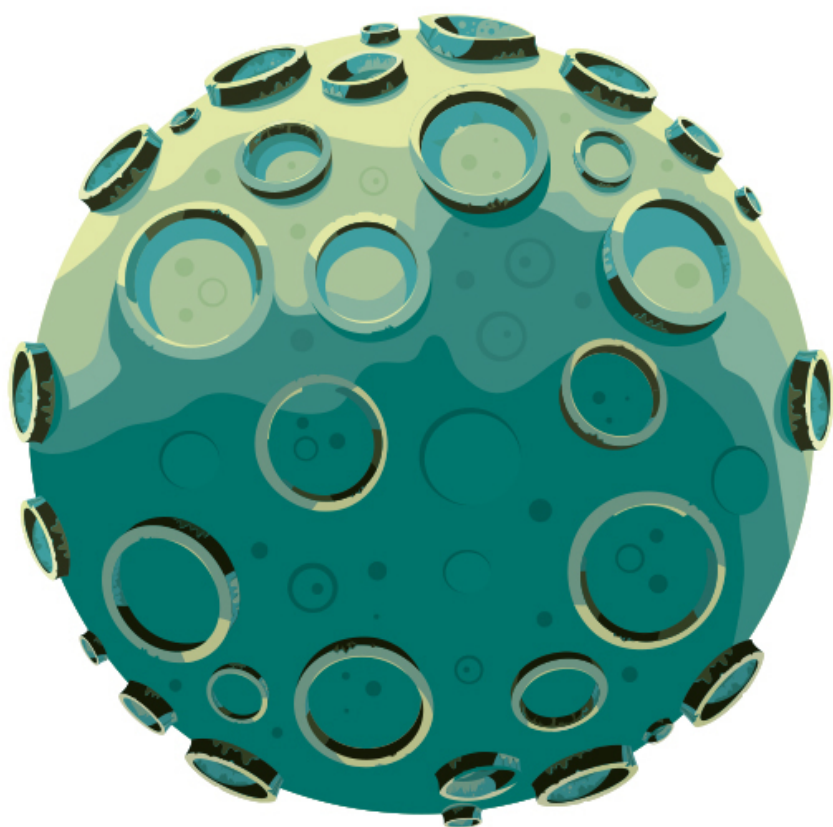
Meteor

Criando aplicações web realtime com JavaScript

CAIO RIBEIRO PEREIRA

Agradecimentos

Obrigado, Deus, por mais uma conquista em minha vida! Obrigado pai e mãe pelo amor, força e incentivo em todas as etapas decisivas de minha vida. Obrigado por tudo e principalmente por estar ao meu lado em todos os momentos.

Um agradecimento especial à minha namorada Natália Santos, obrigado pelo apoio, compreensão e companhia, sei que em alguns momentos não foi fácil para nós, mas o importante é que conseguimos juntos, e essa vitória eu dedico totalmente para você e minha família.

Agradeço à sra. Charlotte Bento de Carvalho, pelo apoio e incentivo nos meus estudos desde a escola até a minha formatura na faculdade.

Obrigado ao pessoal da editora Casa do Código, em especial ao Paulo Silveira e Adriano Almeida. Muito obrigado pelo suporte, apoio, pela confiança e por mais esta nova oportunidade.

Obrigado à galera da comunidade NodeBR e Meteor Brasil, Os *feedbacks* de vocês ajudaram muito a melhorar este livro e também meu livro anterior de Node.js.

Obrigado também aos leitores do blog meu Underground WebDev (http://udgwebdev.com) que acompanham e comentam os posts frequentemente.

Por último, obrigado a você, prezado leitor, por adquirir meu novo livro. Que este seja, uma boa referência de estudos sobre Meteor.

Sobre o autor

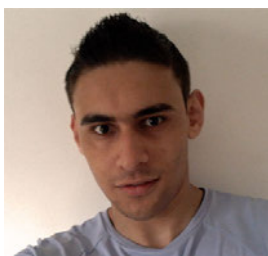



Figura 1: Caio Ribeiro Pereira

Caio Ribeiro Pereira trabalha como Web Developer na BankFacil, bacharel em Sistemas de Informação pela Universidade Católica de Santos, sua experiência é baseada no domínio dessa sopa de letrinhas: Node.js, Meteor, Javascript, Ruby, Java, LevelDB, MongoDB, Redis, Filosofia Lean, Scrum, XP e TDD.

Blogueiro nos tempos livres, apaixonado por desenvolvimento de software, web, tecnologias, filmes e seriados. Participante ativo das comunidades:

- NodeBR: Comunidade Brasileira de Node.js
- MeteorBrasil: Comunidade Brasileira de Meteor
- DevInSantos: Grupo de Desenvolvedores de Software em Santos

Iniciou em 2011 como palestrante nos eventos DevInSantos e Exatec, abordando temas atuais sobre Node.js e Javascript. Criador do Node Web

Modules (um site que apresenta dados estatísticos sobre os frameworks web para Node.js) e fundador do Google Groups Meteor Brasil.

Autor dos Blogs: Underground WebDev e Underground Linux.

Prefácio

As mudanças do mundo web

Atualmente, o JavaScript já é uma opção real e aceitável no desenvolvimento server-side em muitos projetos. Isso traz como uma das vantagens a possibilidade de construir uma aplicação utilizando apenas uma única linguagem, tanto no cliente (através do clássico JavaScript compatível em todos os *browsers*) como no servidor (através do íncrivel e poderoso Node.js!).

Graças ao Node.js, diversos frameworks web surgiram — diferente das outras linguagens, existem mais de 30 frameworks web para Node.js! Não acredita? Então veja com seus próprios olhos essa lista de frameworks acessando o site:

http://nodewebmodules.com

Express e Meteor são os frameworks que mais se destacam, ambos são frameworks para Node.js com características únicas. O Express é considerado um framework minimalista, focado em criar projetos através de estruturas customizadas pelo desenvolvedor. Com ele é possível criar serviços REST, aplicações web tanto em padrão MVC *(Model-View-Controller)* como MVR *(Model-View-Routes)* ou totalmente sem padrão em um único arquivo, tudo vai depender das boas práticas aplicadas pelo desenvolvedor. Uma aplicação Express pode ser pequena ou de grande porte, tudo vai depender dos requisitos e, principalmente, de como vai organizar todos códigos nele, afinal com o Express você tem o poder de aplicar suas próprias convenções e organizações de código. Outro detalhe é que você utiliza seu próprio framework de persistência de dados, então todo controle fica em suas mãos.

A princípio, se você possui um bom conhecimento sobre o Express e os demais frameworks necessários para construir sua aplicação, então você terá velocidade suficiente para desenvolver uma aplicação de forma ágil. Mas e

se existisse um framework Node.js, cujo foco é prototipar telas em extrema velocidade? Tudo isso utilizando componentes prontos que são facilmente customizáveis e com configurações complexas simplificadas alto nível para você utilizar (aplicando os princípios de convenção sobre configuração). No mundo Ruby, isso deu certo através do inovador framework Rails; agora temos uma inovação semelhante no mundo Node.js, essa inovação se chama Meteor.

A quem se destina este livro?

Esse livro é destinado aos desenvolvedores que tenham pelo menos conhecimentos básicos de Node.js, Javascript e arquitetura web cliente-servidor. Ter domínio desses conceitos, mesmo que a nível básico, será necessário para que a leitura seja de fácil entendimento.

Como devo estudar?

Este livro é praticamente um *hands-on* que visa ensinar os principais conceitos do Meteor através da construção de uma aplicação web do zero. Ao decorrer da leitura serão apresentados diversos conceitos e muito código para implementar no projeto, aplicar na prática os conceitos teóricos e aprender boas práticas e convenções desta tecnologia.

Sumário

Versão: 17.2.5

Capítulo 1

Introdução

1.1 Conhecendo seu mundo

O Meteor é um framework web full-stack 100% JavaScript, ou seja, com ele você vai construir aplicações programando em todas as camadas: cliente, servidor e banco de dados, usando JavaScript, Node.js e MongoDB, tudo isso utilizando apenas uma única linguagem de programação, o JavaScript.

Outro detalhe importante é que o foco desse framework é a prototipagem rápida, fazendo com que trabalhos que levariam meses sejam realizados em semanas, ou até mesmo alguns dias. Isso é possível graças aos seus recursos que visam automatizar tarefas repetitivas, exibir respostas imediatas no browser, diversas convenções, vários componentes customizáveis prontos para uso e também as configurações complexas de baixo nível que estão simplificadas para o desenvolvedor. Por *default*, suas aplicações Meteor serão em formato *single-page real-time*, mas também é possível criar aplicações *multi-page*

orientado a rotas. Este framework é considerado como um **MVVM** *(Model-View View-Model)*, ou seja, não existe *controllers* ou *routes* por default, mas é customizá-lo com o propósito de adicionar *controllers*, *routes* e outros *patterns* através da inclusão *packages third-party*.

O Meteor é uma plataforma de desenvolvimento completa, sendo que o mecanismo principal responsável por toda magia *back-end* é o Node.js. Assim como o Node.js possui seu gerenciador de módulos — o NPM (*Node Package Manager*), o Meteor possui o seu próprio gerenciador de *packages* (sim, módulos e packages são a mesma coisa, ou seja, são frameworks), que se chama **Atmosphere**. Este permite usar projetos *third-party* criados pela comunidade, sem contar que também é possível utilizar a maioria — mas não todos — dos módulos NPM dentro de uma aplicação Meteor (algo que será explicado nos capítulos futuros). Isso pode evitar a "invenção da roda", porém tudo vai depender da compatibilidade de tal módulo Node no contexto do Meteor.

Trabalhar com Meteor é trabalhar com JavaScript, e isso faz com que muitos desenvolvedores tenham uma curva de aprendizado rápida, e em poucos dias de estudos você terá dominado os principais pontos e aspectos deste framework. Se você não conhece Node.js mas domina JavaScript, não tenha receio de ler esse livro, pois poucas coisas sobre Node.js serão abordadas aqui.

Este é um framework de muitos recursos, ou seja, constituído por um conjunto de frameworks do mundo Node.js e JavaScript. Algumas bibliotecas conhecidas que fazem parte dele são:

- SockJS — framework emulador de WebSockets e responsável pelo funcionamento do protocolo DDP (*Data Distribution Protocol*).
- MongoDB — banco de dados default.
- Handlebars — template engine.
- PubSub — biblioteca de emissão e escuta de eventos via pattern: *publisher / subscriber*.
- MiniMongo — API *client-side* que possui a maioria das funcionalidades do MongoDB.

- Connect — módulo Node.js com funcionalidades para trabalhar com protocolo HTTP.

Esses são alguns frameworks internos que são essenciais para dar vida ao Meteor.

1.2 Os 7 princípios do Meteor

O Meteor é um framework totalmente inovador, ele engloba diversas boas práticas de frameworks como Rails do Ruby, Django do Python, Express do Node.js, além da própria equipe adotar suas próprias convenções. Tudo isso surgiu baseado em 7 princípios, e eles são:

1 Data on the wire

Figura 1.1: Handlebars é o template engine do Meteor.

Não envie HTML pela rede e sim apenas dados, deixando que o cliente

decida como apresentá-los.

2 One language

Figura 1.2: Node.js, Javascript e MongoDB.

Escreva código JavaScript em todas as camadas: cliente, servidor e banco de dados. Isso simplifica e agiliza o desenvolvimento, além de garantir uma curva de aprendizado baixa.

3 Database anywhere

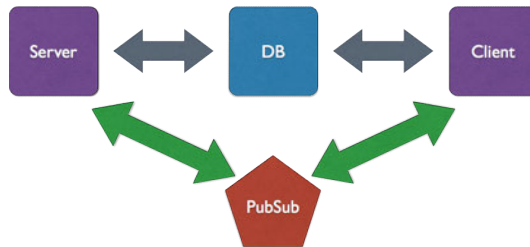



Figura 1.3: Banco de dados compartilhado entre cliente e servidor.

Utilize uma API de interface única e transparente que lhe permite acessar o banco de dados tanto no cliente como no servidor.

4 Latency compensation

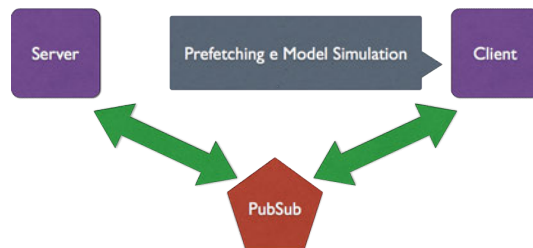



Figura 1.4: Latência zero ao acessar dados via cliente.

No cliente é usado *prefetching* e *model simulation* na API client-side do banco de dados para atingir latência zero no acesso de seus recursos.

5 Full-Stack Reactivity

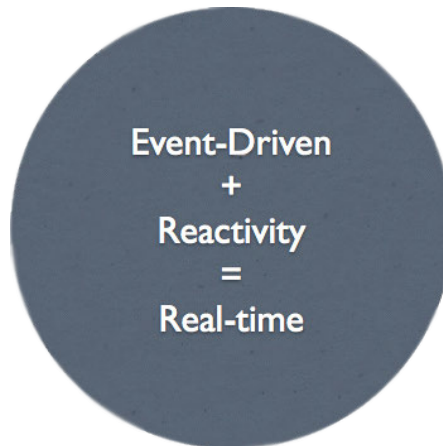



Figura 1.5: Meteor é um framework full-stack real-time.

Por default tudo funciona em *real-time*. E todas as camadas da aplicação adotam o paradigma orientado a eventos, que é herdado do Node.js.

6 Embrace the ecosystem

Figura 1.6: Meteor é mais uma iniciativa open-source.

Totalmente open-source, o Meteor possui suas convenções pelas quais agrega novos conceitos e valores, em vez de ser uma ferramenta que substituirá outros frameworks.

7 Simplicity equals Productivity

Figura 1.7: Aumente sua produtividade trabalhando com Meteor!

Seja produtivo! Desenvolva de mais *features* de forma rápida e simplificada. O Meteor mantém um conjunto de APIs fáceis de implementar e a comunidade Meteor está sempre colaborando para evolução do framework.

Capítulo 2

Configurando o ambiente de desenvolvimento

2.1 Detalhes sobre a instalação

Para instalar o Meteor, primeiro temos que ter o Node.js e MongoDB instalados. Nesta seção explicarei como instalar e configurar o ambiente de desenvolvimento de cada dependência do Meteor.

2.2 Node.js

Instalação

Para configurar o ambiente Node.js, independente de qual sistema operacional você utilizar, as dicas serão as mesmas. É claro que os procedimentos

serão diferentes para cada sistema (principalmente para o Windows, mas não será nada grave).

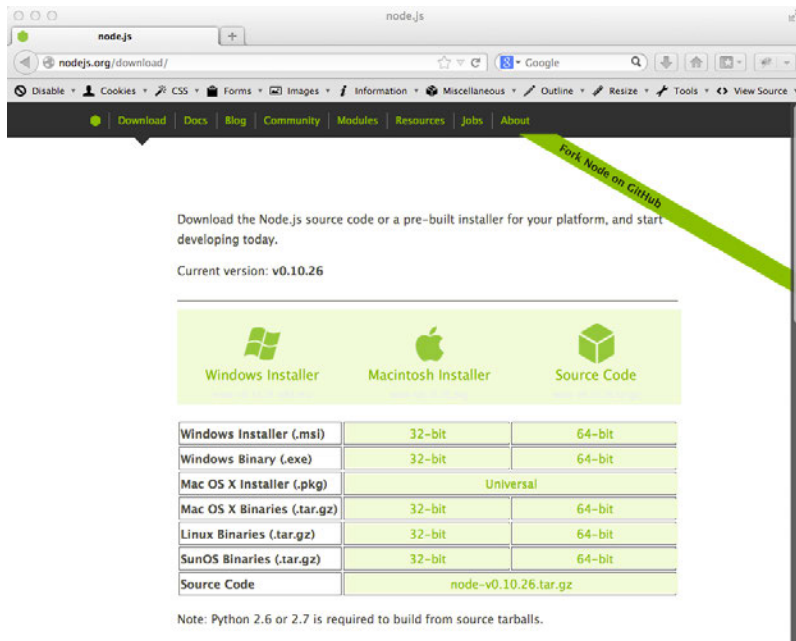



Figura 2.1: Página de Download do Node.js.

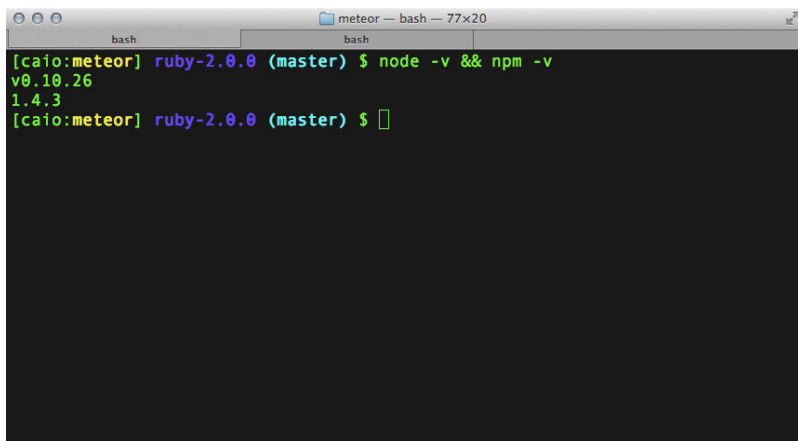
Instalando Node.js: primeiro passo, acesse o site oficial: (http://nodejs.org) e clique em `Download`. Para usuários do Windows e MacOSX, basta baixar os seus instaladores e executá-los normalmente. Para quem já utiliza Linux com *Package Manager* instalado, acesse esse link (https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager) , que é referente às instruções sobre como instalá-lo em diferentes sistemas. Instale o Node.js de acordo com seu sistema e, caso não ocorra problemas, basta abrir o seu terminal console ou prompt de comando e digitar o comando: `node -v && npm -v` para ver as respectivas versões do Node.js e NPM (*Node Package Manager*) que foram instaladas.

```
meteor — bash — 77×20
bash    bash
[caio:meteor] ruby-2.0.0 (master) $ node -v && npm -v
v0.10.26
1.4.3
[caio:meteor] ruby-2.0.0 (master) $
```

Figura 2.2: Versão do Node.js e NPM utilizada neste livro.

A última versão estável utilizada neste livro é **Node 0.10.26**, junto do **NPM 1.4.3**.

> **A importância de instalar versões recentes do Node.js**
>
> É altamente recomendável utilizar uma versão igual ou superior à **v0.10.22**, pois recentemente foram identificadas algumas vulnerabilidades que permitiam ataque *DoS* via HTTP em versões anteriores. Veja mais detalhes no blog oficial: http://blog.nodejs.org/2013/10/22/cve-2013-4450-http-server-pipeline-flood-dos

Configuração do ambiente

Para configurá-lo, basta adicionar uma variável de ambiente, o `NODE_ENV` com valor "development". Em sistemas Linux ou OSX, basta acessar o arquivo `.bash_profile` ou `.bashrc` com um editor de texto qualquer, em modo super user (**sudo**), e adicionar o seguinte comando: `export NODE_ENV='development'`.

No Windows 7, o processo é um pouco diferente.

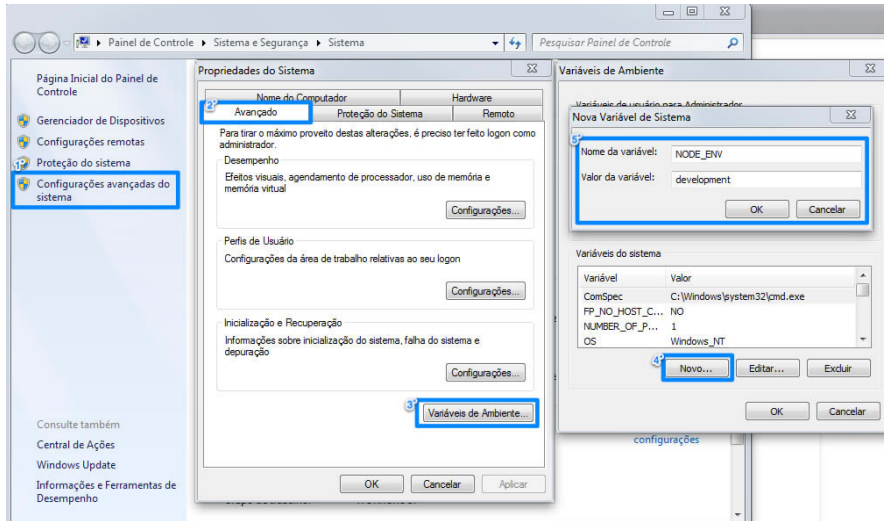



Figura 2.3: Configurando a variável NODE_ENV no Windows 7.

Clique com botão direito no ícone `Meu Computador` e selecione a opção `Propriedades`. No lado esquerdo da janela, clique no link `Configurações avançadas do sistema`. Na janela seguinte, acesse a aba `Avançado` e clique no botão `Variáveis de Ambiente...`; agora no campo `Variáveis do sistema` clique no botão `Novo...`, em "nome da variável" digite `NODE_ENV` e em valor da variável" digite `development`. Após finalizar essa tarefa, reinicie seu computador para carregar essa variável no sistema operacional.

Rodando o Node.js

Para testarmos o ambiente, executaremos o nosso primeiro programa *Hello World*. Execute o comando: `node` para acessarmos seu modo REPL (*Read-Eval-Print-Loop*), que permite executar código JavaScript diretamente no terminal. Digite `console.log("Hello World");` e tecle `ENTER` para executá-lo na hora.

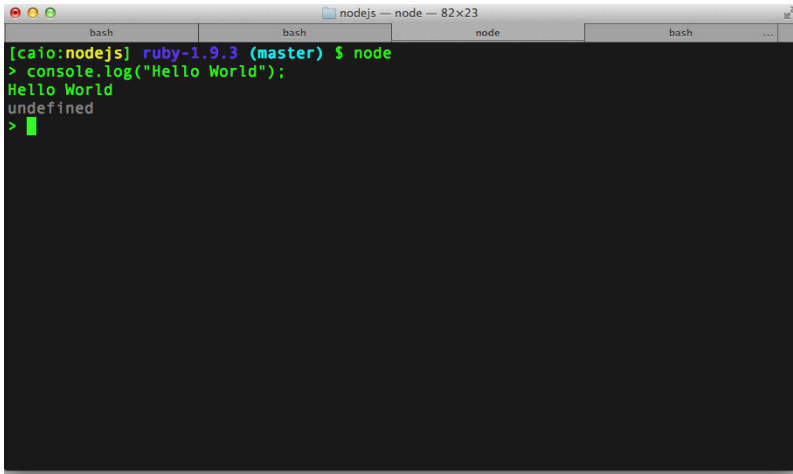



Figura 2.4: Hello World via REPL do Node.js

2.3 MongoDB

Instalação

Com Node.js funcionando, falta instalar mais uma dependência importante do Meteor, o MongoDB. Não há segredos para instalá-lo, tanto é que utilizaremos suas configurações padrões de ambiente *development*. Vamos lá! Acesse este link: http://www.mongodb.org/downloads e faça o download do MongoDB compatível com seu sistema operacional.

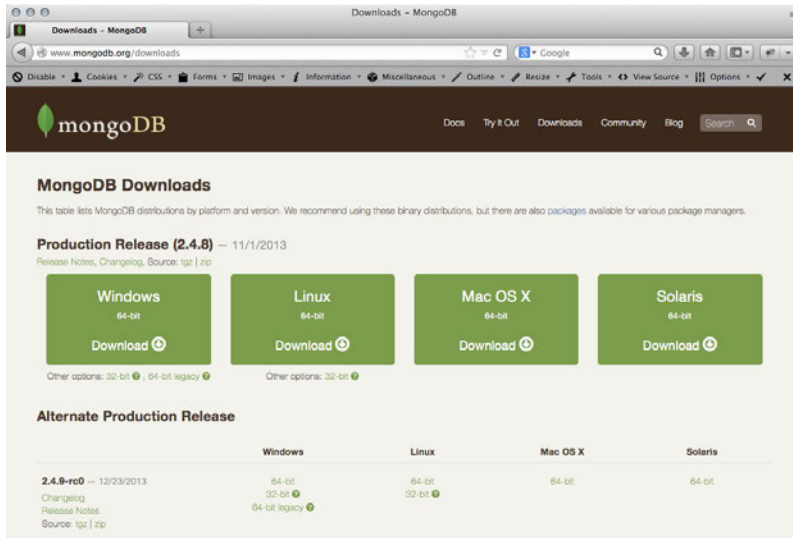



Figura 2.5: Página de download do MongoDB.

Neste livro, utilizaremos sua última versão estável: **2.4.8**. O MongoDB será instalado via HomeBrew (http://mxcl.github.com/homebrew) , que é um gerenciador de pacotes para Mac. Para instalá-lo execute os comandos a seguir:

```
brew update
brew install mongodb
```

Se você estiver no Linux Ubuntu, terá um pouco mais de trabalho, porém é possível instalá-lo via comando `apt-get`. Primeiro em modo `sudo` (super usuário), edite o arquivo `/etc/apt/sources.list.d/mongodb.list`, adicionando no final do arquivo o comando:

```
deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart
    dist 10gen
```

Salve-o e execute os próximos comandos:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80
                 --recv 7F0CEB10
```

```
sudo apt-get update
sudo apt-get install mongodb-10gen
```

Já no Windows, o processo é tão mais trabalhoso que nem apresentarei neste livro...brincadeira! Instalar no Windows também é fácil, apenas baixe o MongoDB, crie as pastas `C:\data\db` e `C:\mongodb`, e descompacte o conteúdo do arquivo `zip` do MongoDB dentro de `C:\mongodb`.

> **Sistema operacional utilizado neste livro**
>
> Todos os frameworks apresentados neste livro serão trabalhados em um sistema MacOSX da *Apple*, mas sinta-se à vontade para trabalhar em outros sistemas como Linux ou Unix. Somente no Windows você sofrerá alguns problemas mas, ainda neste ano de 2014, será lançada uma versão com várias otimizações que irão compatibilizar com Windows.

Configuração do ambiente

Para executar o MongoDB, precisamos configurá-lo como serviço no sistema. Para isso, abra o terminal ou prompt de comando e siga as instruções a seguir de acordo com seu sistema operacional:

- **MacOSX**: já vem configurado como serviço através do homebrew.
- **Linux Ubuntu**: já vem configurado como serviço através do apt-get.
- **Windows**: acesse esse link http://docs.mongodb.org/manual/tutorial/install-mongodb-on-windows e siga passo a passo para configurá-lo corretamente no Windows, pois há variações de configuração entre uma **versão 32bits e 64bits do Windows**.

Rodando o MongoDB

Agora que temos o MongoDB configurado, podemos usar seus comandos no terminal.

Para acessar seu CLI, execute o comando: `mongo`

O MongoDB cria *databases* de forma dinâmica, ou seja, ao usar o comando `use nome_do_database`, automaticamente ele cria um novo ou acessa um existente database. O mesmo conceito é aplicado em suas *collections* e atributos de suas collections.

Faça o teste você mesmo! Execute essa sequência de comandos:

```
use db_test;
db.users.insert({nome: "MongoDB", version: "2.4.8"});
db.users.insert({nome: "CouchDB", version: "1.5.0"});
db.users.insert({nome: "LevelDB", version: "1.1"});
db.users.insert({nome: "MariaDB", version: "10.0.7"});
db.users.find();
```

2.4 Instalando o Meteor

Agora vamos ao que interessa! Instalar e configurar o Meteor em sua máquina. A versão do Meteor usada neste livro é a **0.8.0**.

Instalação em ambiente Mac/Unix/Linux

Instalar o Meteor é super simples, apenas execute:

```
curl https://install.meteor.com | /bin/sh
```

Instalação em ambiente Windows

Somente no Windows existem algumas barreiras para instalar o Meteor, mas tudo isso será resolvido na futura **versão 1.0.0**. Enquanto isso, recomendo que você instale uma máquina virtual rodando Linux e siga as instruções de instalação anterior para prosseguir no livro.

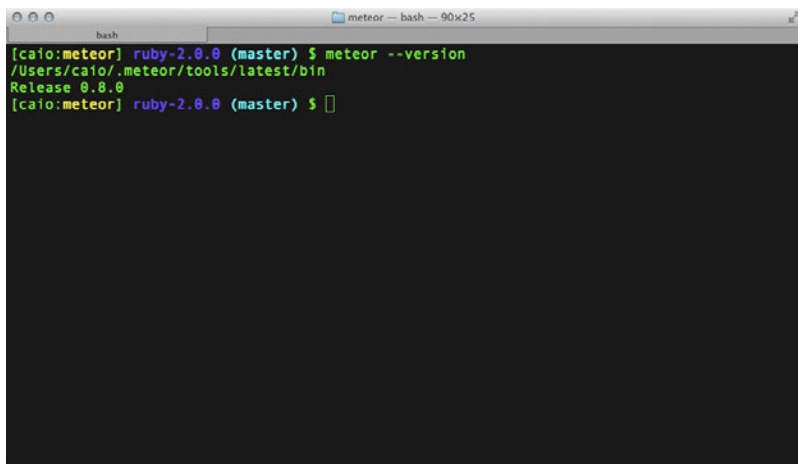



Figura 2.6: Versão do Meteor usada neste livro

2.5 Rodando o Meteor

Que tal criarmos nosso app de teste? Para mostrar o quão fácil é fazer esta magia, apenas execute esses 3 comandos:

```
meteor create meu-app-teste
cd meu-app-teste
meteor
```

Por default, o Meteor cria um aplicativo de "Hello World" sem a necessidade de você codificá-lo. Seu comando *scaffold* gerou 3 arquivos (1 html, 1 css e 1 js) com o mesmo nome `meu-app-teste`. O arquivo `meu-app-teste.html` veio com o seguinte código:

```
<head>
 <title>meu-app-teste</title>
</head>

<body>
 {{> hello}}
</body>
```

```
<template name="hello">
 <h1>Hello World!</h1>
 {{greeting}}
 <input type="button" value="Click" />
</template>
```

Repare na tag `<template name="hello">`: é uma tag especial do Handlebars, cujo conteúdo é dinamicamente injetado dentro da tag `<body>` através da marcação `{{> hello}}`. Caso você exclua essa marcação, o template não será renderizado. Quer fazer um teste? Primeiro acesse em seu browser o endereço http://localhost:3000. Agora exclua a marcação `{{> hello}}`. Repare que, instantaneamente, o Meteor atualizou o HTML no browser, afinal ele possui o mecanismo muito produtivo, conhecido pelo nome de *Hot-Pushes*. Ele faz *auto-reload* no navegador ao alterar qualquer código do projeto. Ao trabalhar com **CSS**, você na hora realiza as mudanças do layout, como por exemplo, edite o `meu-app-teste.css` inserindo o conteúdo a seguir:

```
body {
 margin: 0 auto;
 text-align: center;
}
h1 {
 font-size: 4em;
}
```

Para finalizar, temos também o `meu-app-teste.js`, um código que utiliza funções de cliente e servidor. Nele colocamos códigos a serem executado no cliente dentro da condicional `Meteor.isClient`, e o que tiver que ser executado no servidor colocamos dentro de `Meteor.isServer`. Um detalhe importante é que todos os códigos que estiverem fora dessas duas condicionais serão carregados em ambas camadas. Por default, Meteor gerou o seguinte código JavaScript:

```
if (Meteor.isClient) {
  Template.hello.greeting = function () {
    return "Welcome to meu-app-teste.";
```

```
  };

  Template.hello.events({
    'click input' : function () {
      // template data, if any, is available in 'this'
      if (typeof console !== 'undefined')
        console.log("You pressed the button");
    }
  });
}

if (Meteor.isServer) {
  Meteor.startup(function () {
    // code to run on server at startup
  });
}
```

Dentro de `Meteor.isClient` temos uma declaração da função `Template.hello.greeting`, referente ao `{{greeting}}` que está dentro da tag `<template name="hello">`. Esta é uma das convenções do Meteor em que você deve declarar funções para templates utilizando a convenção `Template.nome_do_template.nome_da_marcação`. Outro código importante no cliente foi a criação de eventos para o mesmo template. Você sempre trabalhará com eventos de forma imperativa no Meteor, semelhante ao framework Backbone (http://backbonejs.org) . Para criar um evento, basta seguir a convenção `Template.nome_do_template.events` e declarar um hash de eventos, cuja chave é uma *string* referente ao "evento seletor" e o valor é uma função JavaScript.

Em `Meteor.isServer`, temos a função principal de startup do projeto, o `Meteor.startup`, cujo callback não possui código. Essa função também possui o mesmo comportamento ao ser declarada no lado do cliente. Vamos ver o `Meteor.startup` em ação? Edite o `meu-app-teste.js` e faça as seguintes alterações:

```
if (Meteor.isClient) {
 Meteor.startup(function () {
```

```
  alert("Iniciando Cliente Meteor.");
 });

 Template.hello.greeting = function () {
  return "Welcome to meu-app-teste.";
 };

 Template.hello.events({
  'click input' : function () {
  // template data, if any, is available in 'this'
  if (typeof console !== 'undefined')
   console.log("You pressed the button");
  }
 });
}

if (Meteor.isServer) {
 Meteor.startup(function () {
  console.log("Iniciando Servidor Meteor.");
 });
}
```

E viu as novas mensagens geradas no servidor e cliente? Se não funcionou, atualize seu browser.

De qualquer forma, temos um grande problema ao trabalhar com essa estrutura gerada pelo Meteor. Dificilmente um projeto grande sobreviveria mantendo todo seu *core* dentro de um único arquivo `.js`, mesmo utilizando inúmeras vezes as variáveis globais `Meteor.isClient` e `Meteor.isServer`. Com o crescimento do projeto, é extremamente importante manter organizada e escalável sua estrutura de pastas e códigos. Mas fique tranquilo, no próximo capítulo este mistério será desvendado!

2.6 Fazendo deploy para testes

Se você é um desenvolvedor que curte publicar com frequência novos updates da aplicação em um ambiente *beta* ou *staging*, o Meteor disponibiliza gratuitamente uma máquina nas nuvens para fazer *deploy* em subdomínio do

aplicacao.meteor.com. Esta é uma máquina com configuração fraca, usada apenas para testar sua aplicação ou para publicar sistemas de pequeno porte. Ela já vem com Node.js, MongoDB e, principalmente, o Meteor. Nos capítulos futuros falarei em mais detalhes sobre *deploying* em ambientes de produção, mas por enquanto explicarei apenas como fazer no meteor.com. Siga os comandos:

```
meteor deploy meu-app-teste --password=senha123
```

Obs.: Caso você não consiga fazer deploy com o nome: meu-app-teste, tente um outro nome qualquer, pois provavelmente o primeiro leitor deste livro já rodou este comando e conseguiu registrar-se com esse nome.

Também é permitido, neste ambiente, fazer redirecionamento de DNS para o seu próprio domínio. Para isso, apenas configure seu serviço DNS apontando um CNAME para origin.meteor.com. Em seguida, faça o deploy do seu projeto utilizando o seu domínio:

```
meteor deploy www.meuapp.com --password=senha123
```

2.7 Gerenciando packages com Meteorite

Como adicional, instalaremos o Meteorite, afinal no decorrer deste livro utilizaremos *packages third-party* que são hospedados no Atmosphere. Não utilizaremos o comando meteor e sim o mrt do Meteorite que além de possuir seus comandos específicos também executa todos os comandos do Meteor.

Como funciona o Meteorite?

O Meteorite possui um comportamento muito parecido com o Gems do Ruby e o NPM do Node.js. As instalações dos pacotes são gerenciadas através do arquivo smart.json, que é criado na raiz do projeto. Também é criado o smart.lock, responsável por manter informações sobre as atuais versões de cada dependência do seu projeto, bem como a versão atual do Meteor. Ao apagar esse arquivo, você força o Meteorite a reinstalar tudo (dependências e a última versão do Meteor), quando você executar, na próxima vez, um desses comandos: mrt, mrt install ou mrt update.

Como receita de bolo, veja a seguir os principais comandos do Meteorite:

- `mrt` — é um alias do comando meteor; sua única diferença é que, antes de iniciar o servidor, ele instala ou atualiza todas as dependências do projeto.
- `mrt create nome-do-app` — é também um alias do comando meteor create, cujo diferencial é que ele cria o arquivo `smart.json`, o qual explicarei no final deste capítulo.
- `mrt add nome-do-package` — é um alias do comando meteor add; sua diferença é que ele instala os pacotes através do Atmosphere. Ele também cria e mantém atualizado o arquivo `smart.json`, que se encontra no diretório raiz do projeto.
- `mrt add nome-do-package --version 0.0.1` — instala um pacote de uma específica versão.
- `mrt install` — instala todos pacotes que estão listado no `smart.json`.
- `mrt update` — atualiza a versão do meteor e de todos os pacotes do projeto.

O arquivo `smart.json`, além de manter os packages dependentes do projeto, também é um descritor do próprio projeto, o qual ele considera como um package. Veja a seguir a estrutura básica de um arquivo `smart.json`:

```
{
  "name": "nome-do-projeto",
  "description": "Descrição do projeto",
  "author": "Usuário <usuario@mail.com>",
  "version": "0.0.1",
  "packages": {
    "nome-do-package-1": {},
    "nome-do-package-2": {}
  }
}
```

Capítulo 3

Criando uma rede social real-time

3.1 Projeto piloto: MeteorBird

Agora que temos o ambiente Meteor instalado e funcionando corretamente, vamos criar nosso projeto piloto, com o qual, no decorrer deste livro, exploraremos os principais recursos do Meteor, bem como explicaremos diversos tópicos e conceitos. O projeto que vamos criar será um rede social, uma versão simplificada do Twitter (http://twitter.com) , que chamaremos de **MeteorBird**.

3.2 Funcionalidades da aplicação

Neste projeto, implementaremos as seguintes *features*:

- Atualizações da timeline em real-time;
- Sign-up através de e-mail e senha;
- Sign-up através de uma conta no Facebook;
- Acessar perfil de um usuário;
- *Follow* e *unfollow* de posts de um usuário;

Sobre o código-fonte deste projeto

Caso queira fazer um *test-drive* antes, para rodá-lo em sua máquina vendo na prática como funciona, faça o download o código-fonte através do meu github pessoal:

https://github.com/caio-ribeiro-pereira/meteor-bird

3.3 Criando o projeto

Nesta seção vamos criar a estrutura desse projeto, isto é, como a estrutura do Meteor é flexível em relação aos diretórios chaves, listarei na sequência quais diretórios serão utilizados e explorados no decorrer deste livro.

Antes de começarmos isso, primeiro vamos criar o projeto. Execute os comandos a seguir para criar o projeto e excluir os arquivos gerados:

```
mrt create meteor-bird
cd meteor-bird
rm meteor-bird.*
```

Agora crie os seguintes diretórios, os quais exploraremos nos capítulos seguintes:

- **client** — diretório de códigos client-side;
- **server** — diretório de códigos server-side;
- **models** — diretório de modelos compartilhados entre cliente e servidor, eles são as *collections* do MongoDB;

- **public/resources** — diretório para incluir arquivos estáticos (utilizaremos para servir imagens);
- **test** — diretório para rodar testes na aplicação.

É claro que no decorrer do caminho criaremos alguns subdiretórios, mas com essa estrutura inicial será mais fácil seguir este livro com uma noção básica sobre as convenções de cada um desses diretórios.

Antes de começar o projeto, utilizaremos o **Twitter Bootstrap 3** (http://getbootstrap.com) como framework CSS para nos auxiliar no desenvolvimento das interface da aplicação. O mais bacana é que o Meteor já possui um package chamado `bootstrap-3` pronto para uso, e para instalá-lo basta rodar o seguinte comando:

```
mrt add bootstrap-3
```

Agora que adicionamos nosso primeiro package, surgiu o arquivo `smart.json` na raíz do projeto. Para deixá-lo mais descritivo, vamos adicionar algumas informações sobre o projeto Meteor Bird. Para isso, edite esse arquivo com base no seguinte conteúdo:

```
{
  "name": "meteor-bird",
  "description": "Meteor Bird Social Network",
  "author": "Caio R. Pereira <caio.ribeiro.pereira@gmail.com>",
  "version": "0.0.1",
  "packages": {
    "bootstrap-3": {}
  }
}
```

Obs.: Mude o nome do campo `author` para o seu nome e e-mail.

Também criaremos um simples arquivo `css` que irá definir uma cor de plano de fundo tamanho real do *container* da aplicação. Crie o arquivo `client/stylesheets/application.css`, inclua e estilize os elementos a seguir:

```
body {
    background-color: #EFEFEF;
}
.container {
    background-color: #FFF;
    width: 780px;
}
```

Com isso já temos uma conjunto de estilos para construir páginas bonitas e amigáveis na aplicação.

Não pare de ler este livro! No próximo capítulo botaremos as mãos na massa!

Capítulo 4

Implementando uma timeline de posts

4.1 Estruturando os templates

Começando o desenvolvimento da nossa rede social, vamos criar os principais templates da aplicação: cabeçalho e rodapé. Todos os templates serão criados na pasta `client/views`, mas somente o template principal fica dentro da pasta `client` e ele deve ser chamado de `client/index.html`. Vamos começar criando o cabeçalho da página no template `client/views/header.html`:

```
<template name="header">
 <header class="container">
  <nav class="navbar navbar-default">
```

```
    <div class="navbar-header">
     <a class="navbar-brand">Meteor Bird</a>
    </div>
   </nav>
  </header>
</template>
```

Em seguida, vamos criar o rodapé em `client/views/footer.html`:

```
<template name="footer">
 <footer class="container">
  <hr>
  <p class="text-center">
   <small>Meteor Bird</small>
  </p>
 </footer>
</template>
```

Com base nesses dois templates, já montamos a estrutura básica do layout. Que tal testar? Levante o servidor Meteor rodando o comando `mrt` e acesse http://localhost:3000 para ver os resultados.

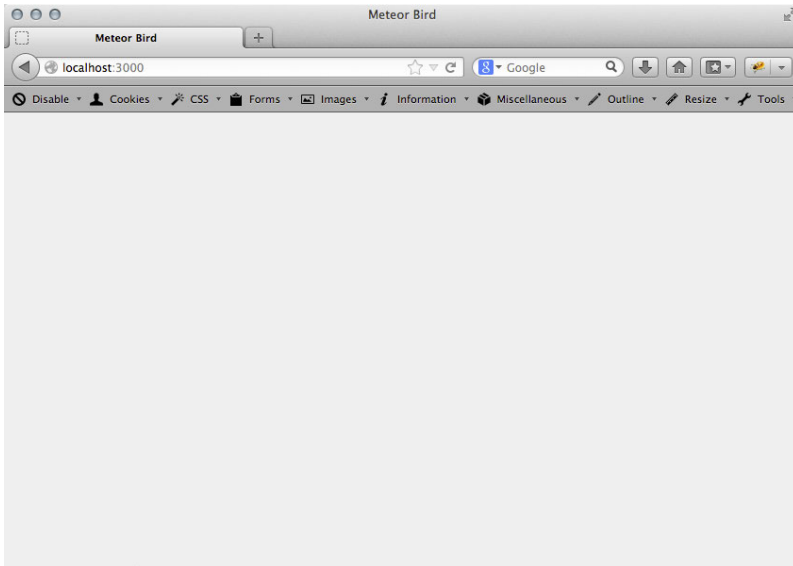



Figura 4.1: Por que a página está em branco?

Mas o que houve? Por que não apareceu nada? Cadê o cabeçalho e rodapé? Calma! Esqueci de explicar um pequeno detalhe: por default, o HTML não renderiza a tag `<template>` e quem trabalha com todo conteúdo dos template no Meteor é o Handlebars. Ele o *Template Engine* responsável por injetar o conteúdo de um template dentro do HTML principal (além de ter outras funcionalidades focadas em manipular conteúdo dinâmico em uma página HTML).

O Meteor abstrai todo código JavaScript referente às funções do Handlebars além de outras funções de manipulação de views através do recente framework introduzido na versão `0.8.x`, chamado Blaze.

https://github.com/meteor/meteor/wiki/Using-Blaze

Graças ao Blaze você não vai precisar compilar um template e chamar funções complexas para renderizá-lo, você apenas vai criar os template no projeto e chamá-lo onde quiser através da marcação `{{> nome_do_template}}`. Incluir um template dentro do HTML principal é muito simples, veja como incluiremos o cabeçalho e rodapé editando o `client/index.html`:

```
<head>
 <title>Meteor Bird</title>
</head>
<body>
 {{> header}}
 {{> footer}}
</body>
```

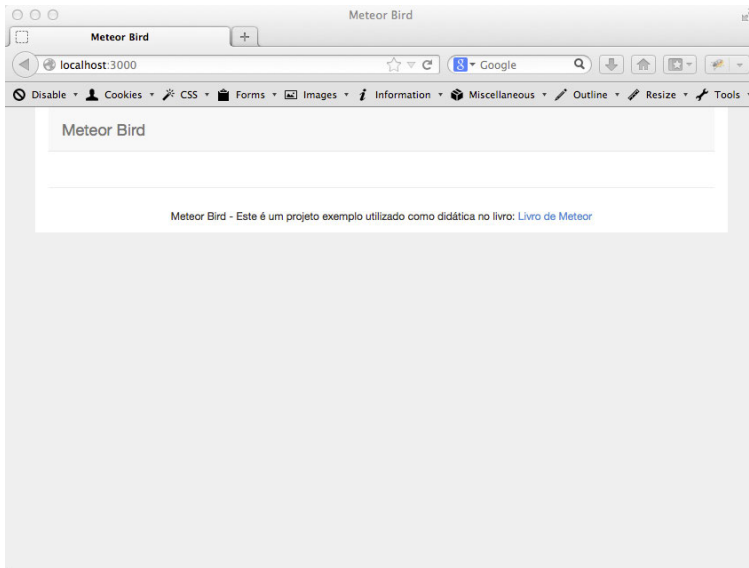



Figura 4.2: Cabeçalho e rodapé na página principal

4.2 Criando o template da timeline

Nesta seção, vamos criar a timeline de posts, que será a primeira *feature* do projeto. A timeline deve mostrar posts do próprio usuário e principalmente posts de seus seguidores, porém **focaremos no momento em apenas exibir os posts do próprio usuário**. Faremos esta implementação em duas etapas, a primeira será somente uma renderização simples da timeline com dados *fakes*, para prototipar seu layout. No decorrer dos capítulos, implementaremos o código final dessa funcionalidade.

Crie o diretório `client/lib/helpers`. Este será um diretório de *helpers*, através do qual criaremos funções com regras de negócio focadas em apresentar algo na *view* do cliente. Nele, podemos utilizar funções das collections, outras estruturas de dados ou até mesmo bibliotecas externas. O objetivo de um helper no Meteor é de criar funções que retornem resultados apresentáveis nas views.

Vamos criar nossa **timeline fake**? Dentro de `client/lib/helpers`, crie o arquivo `timeline.js`, com o seguinte código:

```
Template.timeline.helpers({
 posts: function() {
  return [
   {message: "Ola!"},
   {message: "tudo bem?"}
  ];
 }
});
```

Agora, para alcançar o resultado, temos que criar o template da timeline. Crie a view `client/views/home/timeline.html` com o template a seguir:

```
<template name="timeline">
 <div class="row">
  <fieldset class="col-sm-12">
   <legend>últimos posts</legend>
   {{#each posts}}
    <blockquote class="lead">
     <small>{{message}}</small>
    </blockquote>
    <hr>
   {{else}}
    <p class="lead text-center">
         Nenhum post publicado.
    </p>
   {{/each}}
  </fieldset>
 </div>
</template>
```

Repare nas convenções de *namespace* do Meteor. Criamos o helper `timeline.js`, cujo código segue com o namespace `Template.timeline.helpers`. Ele permite registrar funções helpers para o template `timeline`. Criamos uma função chamada `posts`, cujo objetivo é retornar um array de objetos referentes aos dados de um post. Apenas seguindo as convenções de nomes já estamos aptos a criar um template que renderize essa função helper. Basta apenas criar a tag `<template name="timeline">` e, internamente, utilizar sintaxe do Handlebars que permite rodar um loop do array retornado pelo helper `posts`, utilizando a marcação `{{#each posts}}`.

Se você deixou seu servidor Meteor rodando, repare que nada aconteceu no browser. Isso ocorre porque não adicionamos esse template na view (`client/index.html`), então adicione o seguinte conteúdo:

```
<head>
 <title>Meteor Bird</title>
</head>
<body>
 {{> header}}
 <main class="container">
  <section class="col-sm-12">
   {{> timeline}}
  </section>
 </main>
 {{> footer}}
</body>
```

Agora acesse novamente o browser, ou, se ocorrer algum problema, reinicie o servidor teclando no terminal `CTRL+C` (no Linux ou Windows) ou `Control+C` (no MacOSX). Em seguida, execute o comando `mrt` e veja o resultado:

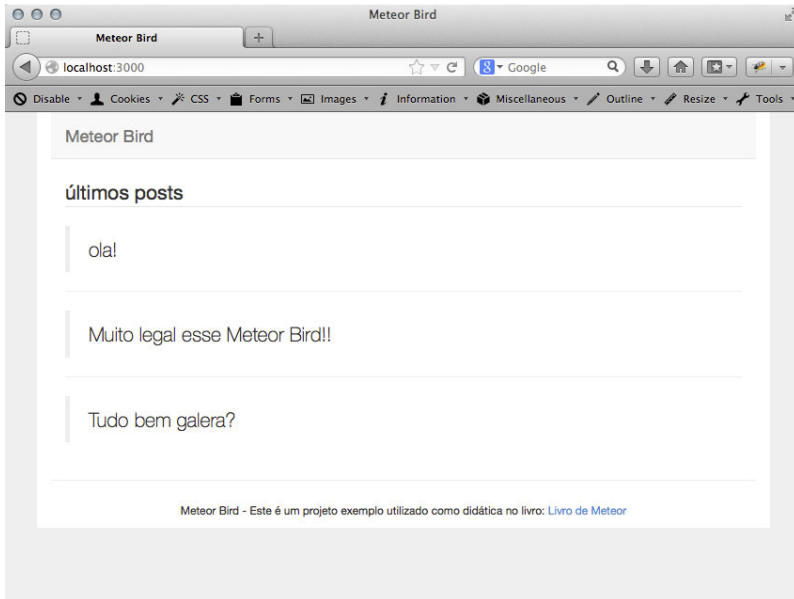



Figura 4.3: Timeline estática.

4.3 Publicando posts na timeline

Essa timeline está estática, que tal criarmos uma caixa de texto permitindo que o usuário publique seus próprios posts? Para isso, utilizaremos um recurso do Meteor, que se chama `Session`.

Diferente das sessions convencionais de outras linguagens, uma `Session` do Meteor não armazena dados na memória do servidor, e sim no cliente. Isso faz com que um simples *refresh* no browser limpe todos os dados do cliente. Outro detalhe importante é que a `Session` utiliza internamente o *pattern reactive*, o principal *pattern* utilizado em muitas classes nativas do Meteor. Sua utilidade é fazer *re-computation*, ou seja, um objeto que adota esse pattern tem a proatividade de reexecutar suas funções de saída toda vez que o estado de seus atributos sofrer alterações.

Por exemplo, o próprio objeto `Session` possui a função de saída `get(chave)`, e como entrada, ele possui a função `set(chave, valor)`.

Todos os pontos da aplicação em que você utilizar `Session.get("X")` serão reexecutados toda vez que um outro ponto da aplicação alterar o valor de `"X"` executando a função `Session.set("X", "novo valor")`. No Meteor, as collections do MongoDB adotam esse pattern e reexecutam sempre que uma collection muda de estado, bem como quando a mesma é excluída do banco.

Esse pattern, como o próprio nome diz, faz com que suas funções trabalhem de forma reativa, ou seja, se mudou algo interno, reexecute sua função de saída referente ao atributo alterado. No quesito produtividade, isso evita que nós, programadores, implementemos essas rotinas de mudança de estado de um objeto, diminuindo incrivelmente muitas linhas de código.

Voltando ao nosso projeto, vamos aplicar o conceito de programação reativa através da implementação da caixa de posts. Primeiro edite o `client/lib/helpers/timeline.js`:

```
Template.timeline.helpers({
 posts: function() {
  return Session.get("posts");
 }
});
```

Com isso, já deixamos nossa timeline preparada para retornar os posts direto de uma `Session`. Agora temos que criar o formulário que permite publicar posts — este será um novo template que incluiremos na view principal (`client/index.html`). Mas antes, vamos criar o template `client/views/home/post.html` com o formulário que segue:

```
<template name="post">
 <div class="row">
  <div class="col-sm-12">
   <form class="form-horizontal">
    <fieldset>
     <legend>novo post</legend>
     <div class="row">
      <div class="col-sm-9">
       <textarea rows="2"
                 class="form-control">
```

```
          </textarea>
        </div>
        <div class="col-sm-1">
          <button class="btn btn-lg">
            Publicar
          </button>
        </div>
      </div>
    </fieldset>
  </form>
 </div>
</div>
</template>
```

Em seguida, vamos adicioná-lo na view principal, editando o `client/index.html`:

```
<head>
 <title>Meteor Bird</title>
</head>
<body>
 {{> header}}
 <main class="container">
  <section class="col-sm-12">
   {{> post}}
   {{> timeline}}
  </section>
 </main>
 {{> footer}}
</body>
```

Para finalizar essa feature, criaremos um *listen* no evento `submit` do formulário, para que capture a mensagem do post e armazene-a em um simples array dentro de `Session.set("posts", posts)`. O Meteor trata os eventos de templates de forma semelhante aos helpers — a diferença é que o nosso evento adotará a convenção `Template.post.events()`. Basicamente, um evento no Meteor é organizado através da estrutura **chave-valor** em que a chave é uma string estruturada por evento e elemento (`"evento`

elemento"), e o valor é uma função em cujo parâmetro retorna o objeto event, semelhante ao framework **jQuery** (http://jquery.com) .

Através do segundo parâmetro, chamado de template, temos acesso às funções template.find e template.findAll, que serão utilizadas para retornar um elemento interno do template. No nosso caso, o template retorna o template atual, e com isso é possível navegar entre seus elementos internos. Utilizaremos a função template.find("textarea") para obter a tag <textarea> e, assim, capturar os dados da mensagem publicada. Em seguida, utilizamos um *shortcut* lógico no trecho var posts = Session.get("posts") || [];, que simplesmente retorna um *array* vazio quando Session.get("posts") retornar undefined. Após obtermos o array de posts, incluiremos um novo post no array e o atualizaremos dentro da Session, via Session.set("posts", posts). Para ver como ficou, crie o arquivo client/lib/events/post.js, seguindo a implementação adiante:

```
Template.post.events({
 "submit form": function(e, template) {
   e.preventDefault();
   var textarea = template.find("textarea");
   var posts = Session.get("posts") || [];
   posts.push({message: textarea.value});
   Session.set("posts", posts);
   textarea.value = "";
 }
});
```

Um detalhe final é que nesta função é obrigatório rodar a função e.preventDefault() para forçar o browser a não realizar uma requisição síncrona no servidor — ou seja, para que ele cancele a submissão padrão do formulário.

Vamos rodar o projeto? Acesse http://localhost:3000 e publique seus posts na timeline.

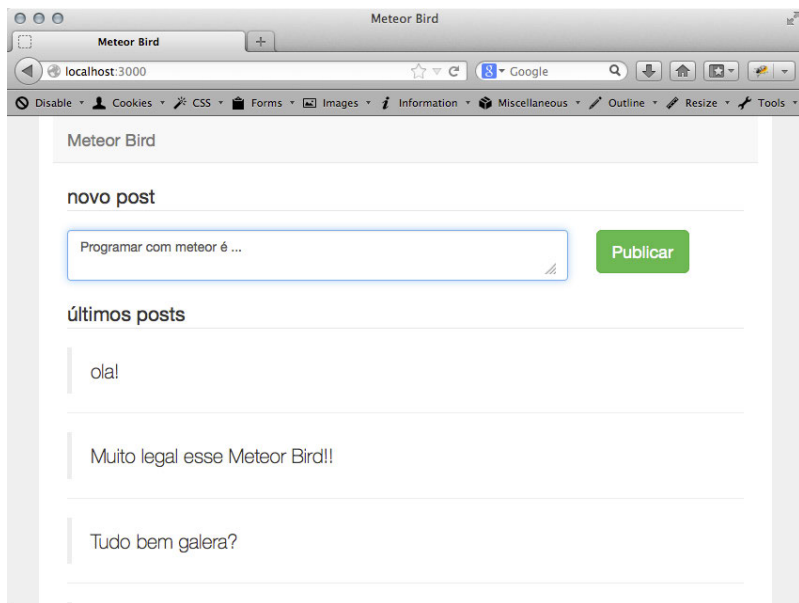



Figura 4.4: Timeline dinâmica!

O mais legal é ver tudo isso funcionando em tempo real! Agora abra um novo browser no mesmo endereço e publique novos posts...

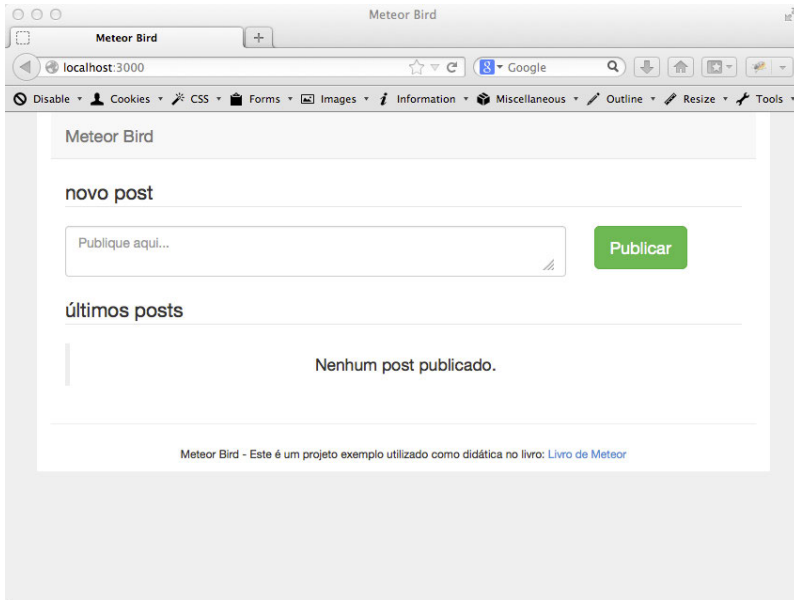



Figura 4.5: Por que a timeline esta vazia?

Mas que estranho! Cadê os posts anteriores da timeline? Esse problema ocorreu devido ao simples fato de que ainda não foi implementado código que envolva o banco de dados! Até agora fizemos uma timeline *client-side* visível apenas para o próprio usuário utilizando `Session`.

4.4 Persistindo e listando posts em tempo-real

Agora que temos o *front-end* de nossa primeira feature, vamos implementar seu *back-end* para torná-lo 100% funcional. Basicamente, criaremos uma collection no MongoDB para armazenar os novos posts que serão listados na timeline em tempo-real.

Começando a implementação, crie o arquivo `models/post.js` com conteúdo a seguir:

```
Post = new Meteor.Collection('posts');
```

Dessa forma, será instanciada a *collection* `Post` em variável global,

permitindo que utilizemos suas funções tanto no cliente como no servidor, afinal o diretório `models` é visível em ambos os lados. Feito isso, faremos uns *refactorings* nos códigos `client/lib/helpers/timeline.js` e `client/lib/events/post.js`, e substituiremos o uso do objeto `Session` pela collection `Post`. Abra e edite `client/lib/helpers/timeline.js`, deixando-o da seguinte forma:

```
Template.timeline.helpers({
 posts: function() {
  return Post.find({});
 }
});
```

Agora modifique também o código `client/lib/events/post.js`:

```
Template.post.events({
 "submit form": function(e, template) {
   e.preventDefault();
   var textarea = template.find("textarea");
   Post.insert({message: textarea.value});
   textarea.value = "";
 }
});
```

Done! Vamos testar a timeline? Reinicie o servidor e acesse http://localhost:3000. Agora todos os posts não serão excluídos e você também poderá acessar essa página em quantas abas quiser e em qualquer navegador ao mesmo tempo.

Parabéns! Terminamos nossa primeira funcionalidade do projeto: implementamos de forma bem simples, porém 100% funcional uma timeline de posts. Continue lendo pois, após criarmos as restantes features, voltaremos nesta timeline para fazer *refactorings* que permitirão a listagem de posts de usuários que você seguir na rede, mas para isso acontecer, vamos criar outras funcionalidades no projeto e faremos a integração delas com a timeline.

Capítulo 5

Signin e Signup de usuários

No capítulo anterior, exploramos o Meteor desenvolvendo o layout e a funcionalidade mínima de uma timeline de posts. Começamos criando os templates, programamos o mínimo de código suficiente para dar vida à timeline e finalizamos o capítulo persistindo e listando os posts no banco de dados MongoDB.

Neste capítulo, vamos aprofundar nossos conhecimentos desenvolvendo a funcionalidade de *Signin* (login) e *Signup* (cadastro), utilizando o conjunto de funcionalidades do package `Accounts`, que é nativa do Meteor.

5.1 Explorando Accounts do Meteor

O `Accounts` possui diversas funções referentes ao processo de *login*, *logout*, recuperação de senhas, mudança de senha, envio de e-mail verificador e cadastro de usuários. Ele é totalmente extensivo e permite a implementação de

autenticações customizáveis através do consumo de APIs externas, como por exemplo autenticação via **Google**, **Facebook**, **Twitter**, **Github**, **Meetup** etc. Diversas variações de `Accounts` são encontradas no Atmosphere:

https://atmospherejs.com/?q=accounts

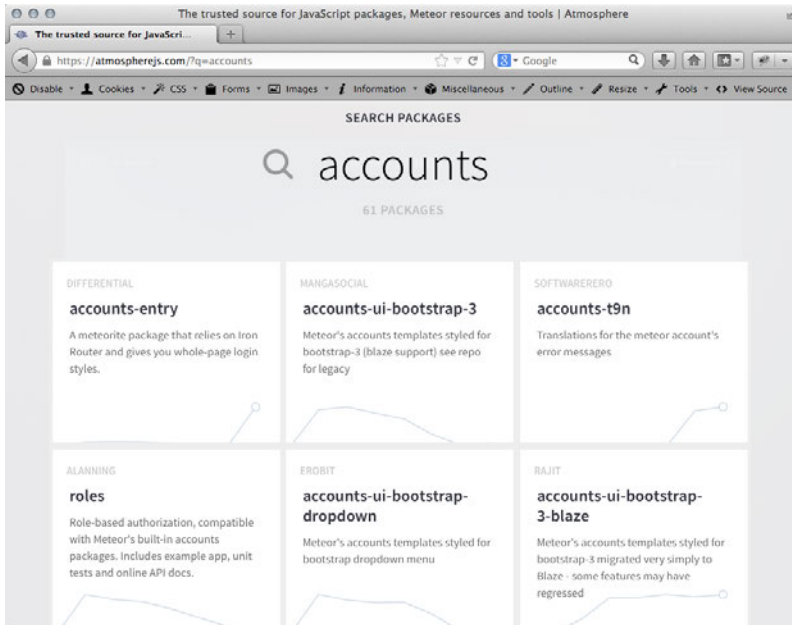



Figura 5.1: Variações do package Accounts.

Com base nessa introdução, vamos desenvolver um simples cadastro e login de usuário diretamente na base local do projeto, e mais para frente implementaremos uma autenticação via Facebook. Por enquanto, adicionaremos três principais `packages` para criar um *signin/signup* convencional:

```
mrt add accounts-base
mrt add accounts-ui-bootstrap-3
mrt add accounts-password
```

O que faz cada package?

- `accounts-base`: API nativa que contém funções essenciais para *signin* e *signup* de usuários, persistindo todos os dados no MongoDB;
- `accounts-password`: este package provê funções para realizar validação, mudança e recuperação de senha, além de possuir funções para envio de e-mail, como e-mail de verificação e *reset* de senha;
- `accounts-ui-bootstrap-3`: possui um conjunto de interfaces prontas para renderizar um formulário de login, cadastro e recuperação de senha, tudo isso utilizando o framework **Twitter Bootstrap 3**.

Agora vamos botar a mão na massa! Edite o template `client/views/header.html`: nele, vamos incluir o *widget* `{{> loginButtons}}` que renderiza um template do `accounts-ui-bootstrap-3`. Este último contém toda interface de login, cadastro e recuperação de conta.

```
<template name="header">
 <header class="container">
  <nav class="navbar navbar-default">
   <div class="navbar-header">
    <a class="navbar-brand">Meteor Bird</a>
   </div>
   <div class="navbar-collapse collapse">
    <ul class="nav navbar-nav navbar-right">
     {{> loginButtons}}
    </ul>
   </div>
  </nav>
 </header>
</template>
```

Veja como ficou legal...

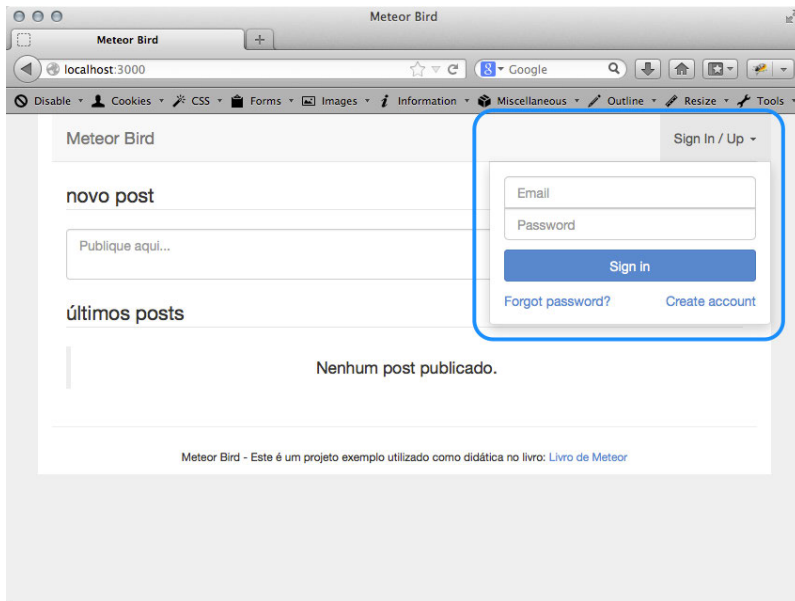



Figura 5.2: Sign In / Up fornecido pelo accounts-ui.

O melhor de tudo isso é que carregamos este widget com apenas uma linha de código e todo o fluxo do *back-end* (cadastro, login e recuperação de senha) já foi fornecido pelo `accounts-base` + `accounts-password`, ou seja, se você cadastrar uma conta, um novo registro na collection `users` será criado. E quando precisar dos dados das collections de usuários, você tem disponível a função nativa `Meteor.users`, que é o mesmo que rodar essa instância: `Users = new Meteor.Collection('users')`.

5.2 Associando posts a um usuário

Nesta etapa, implementaremos um pouco de *back-end*. Vamos associar nossa timeline com o `userId` do usuário. Para fazer isso, utilizaremos o `Meteor.userId()`, que retorna o `id` do usuário da sessão e, em seguida, vamos associá-lo com seus posts publicados. Dessa vez, criaremos o método `Post.publish` em `models/post.js` para abstrair a complexidade dessa funcionalidade. Veja como fazer:

```
Post = new Meteor.Collection('posts');

Post.publish = function(message) {
 this.insert({
  message: message,
  date: new Date(),
  userId: Meteor.userId()
 });
};
```

Consequentemente faremos um *refactoring* em `client/lib/events/post.js`, para utilizar esse novo método `Post.publish()`:

```
Template.post.events({
 "submit form": function(e, template) {
   e.preventDefault();
   var textarea = template.find("textarea");
   Post.publish(textarea.value);
   textarea.value = "";
   return false;
 }
});
```

Para finalizar a associação, vamos criar uma função dentro do modelo `Post`, que será um encapsulamento de `Post.find({userId: userId})`. Ela se chamará `Post.list(userId)` e ficará em `models/post.js`:

```
Post = new Meteor.Collection('posts');

Post.publish = function(message) {
 this.insert({
  message: message,
  date: new Date(),
  userId: Meteor.userId()
 });
};

Post.list = function(userId) {
```

```
 return this.find({userId: userId});
};
```

Para o final, usaremos esta query de listagem para que ele exiba somente posts do usuário logado, utilizando a função `Meteor.userId()` dentro do helper da timeline. Edite o arquivo `client/lib/helpers/timeline.js`:

```
Template.timeline.helpers({
 posts: function() {
  return Post.list(Meteor.userId());
 }
});
```

5.3 Exibindo timeline somente para logados

Agora temos que restringir a timeline para que somente usuários logados publiquem e visualizem os posts. Para resolver isso, existe uma marcação especial do `accounts-base` chamada `{{currentUser}}`, que retorna os dados de um usuário quando o mesmo está logado no sistema. Com isso, criaremos um template pai dentro do diretório `client/views/home` com o nome `home.html`, no qual vamos incorporar os templates `client/views/home/post.html` e `client/views/home/timeline.html`, somente quando o usuário estiver logado na aplicação. Quando não estiver logado o `{{currentUser}}` retornará nulo e com isso será exibida a mensagem: `"Você não esta logado, clique em Sign In / Up."`. Para fazer isso, crie o template `client/views/home/home.html` com as tags a seguir:

```
<template name="home">
 {{#if currentUser}}
  <section class="col-sm-12">
   {{> post}}
   {{> timeline}}
  </section>
 {{else}}
  <div class="text-center">
```

```
    <h1>Meteor Bird</h1>
    <h3>Não esta logado? Clique em "Sign In/Up".</h3>
  </div>
 {{/if}}
</template>
```

Em seguida, vamos atualizar o template principal `client/index.html`, deixando-o mais leve, apenas renderizando os templates `header`, `home` e `footer`.

```
<head>
 <title>Meteor Bird</title>
</head>
<body>
 {{> header}}
 <main class="container">
  {{> home}}
 </main>
 {{> footer}}
</body>
```

Faça o teste! Acesse a URL da aplicação em http://localhost:3000 e visualize as novas telas dedicadas para usuário logado e usuário não-logado.

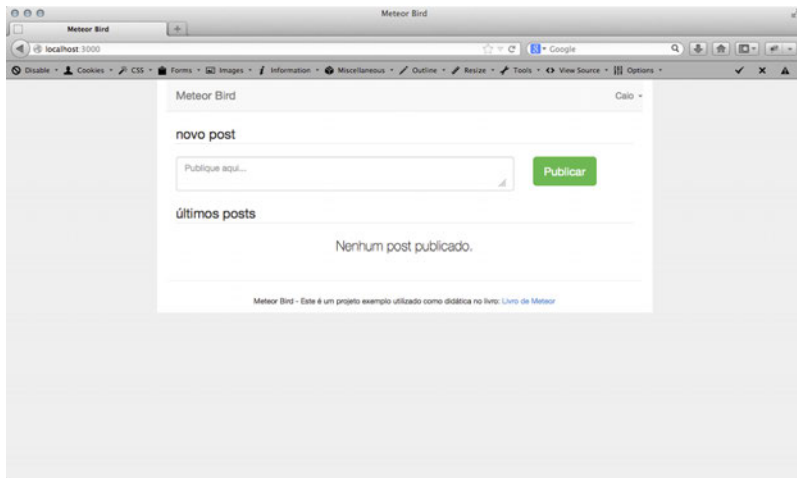



Figura 5.3: Usuário logado.

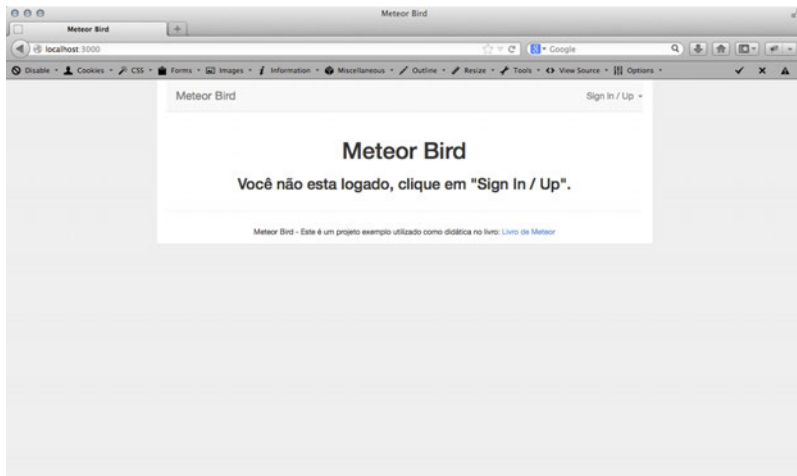



Figura 5.4: Usuário não-logado.

5.4 Autenticação via conta Facebook

O `Accounts` do Meteor é bem extensível, suas interfaces permitem criar variações de *signin* e *signup* que utilizem API de aplicações externas. Existem 6 variações desses packages oferecidos nativamente pelo Meteor e +60 packages *third-party* existentes no **Atmosphere** (https://atmospherejs.com) .

Nesta seção, vamos focar na inclusão de apenas uma variação de `Accounts`. Vamos utilizar o `accounts-facebook` para que os nossos usuários se autentiquem na aplicação através de uma conta Facebook (http://facebook.com) .

Habilitar o `Accounts` do Facebook na aplicação é simples, a parte mais chata é cadastrar uma aplicação no Facebook para gerar as chaves `appId` e `secret` que são usadas para se autenticar na rede social.

Começando a implementação, vamos adicionar o package a seguir:

```
mrt add accounts-facebook
```

Em seguida, adicionaremos uma lista de permissões a respeito dos dados que iremos consumir da conta do usuário quando ele se autenticar em nosso sistema através do Facebook. Quando um usuário permitir seu acesso, ele libera diversas informações de sua conta, como seu e-mail, sua lista de amigos, seus últimos posts, seu perfil e muito mais. Cada informação varia de sistema para sistema, e para conhecer todas as permissões é necessário conhecer a fundo a documentação de sua API (no nosso caso, temos que estudar a API do Facebook através do **Facebook Developers**). Antes de uma aplicação externa permitir tal acesso, é exibida uma página da própria aplicação avisando sobre todas as informações que serão expostas ao se autenticar. No Meteor, utilizamos a função `Accounts.ui.config()` para configurar as permissões de cada autenticação externa via atributo `requestPermissions`. No próximo capítulo explicarei em detalhes como autocompletar o nome do usuário que entrar na rede via Facebook. Porém se você não possui uma conta no Facebook e pretende entrar através do cadastro convencional, então usaremos o atributo `extraSignupFields` para incluir um campo extra, este campo será o nome do usuário. Para aplicar estas configurações na aplicação, crie o arquivo `client/lib/config/accounts_auth.js` com os códigos a seguir:

```
Accounts.ui.config({
 extraSignupFields: [{
  fieldName: 'name',
  fieldLabel: 'Nome'
 }],
 requestPermissions: {
  facebook: ['email', 'user_about_me']
 }
});
```

Para que o cadastro convencional pré-preencha o nome do usuário, teremos que criar um código no servidor, responsável por interceptar os dados do formulário durante o cadastro. Isso se faz através do evento `Accounts.onCreateUser(options, user)`. Nesta função qualquer campo adicional será enviado através do objeto `options['profile']['campo_adicional']`, ou seja, `options['profile']['name']`. Por enquanto vamos apenas atribuir esse campo `name` para o usuário, para isso temos que criar o arquivo `server/lib/accounts.js` e implementar o código abaixo:

```
Accounts.onCreateUser(function(options, user) {
 user['profile'] = options.profile;
 return user;
});
```

Em nossa aplicação, vamos apenas consumir o `email` e `user_about_me` (campos que contém dados básico do perfil — é lá que pegaremos o nome de um usuário Facebook). No *server-side*, temos que definir as chaves para acessar o **serviço do Facebook**, usando as chaves `appId` e `secret`. Para configurar essas chaves vamos utilizar um novo package que gerencia chaves de serviços externos:

```
meteor add service-configuration
```

Com este package, poderemos configurar diversos serviços de forma intuitiva, através das funções do modelo `ServiceConfiguration.configurations`. Para evitar contas duplicadas ou triplicadas de um mesmo serviço, rodamos primeiro a

função `ServiceConfiguration.configurations.remove({})` para apagar todas as chaves dos serviços e, em seguida, rodamos `ServiceConfiguration.configurations.insert()` para registrar cada serviço no banco de dados. No nosso caso, vamos configurar apenas o **serviço de autenticação do Facebook**. Veja como fica essa implementação, crie o `server/lib/services.js` e inclua o código a seguir:

```
ServiceConfiguration.configurations.remove({});
ServiceConfiguration.configurations.insert({
 service: 'facebook',
 appId: 'inserir appId gerado pelo Facebook',
 secret: 'inserir secret gerado pelo Facebook'
});
```

Obs.: Para cadastrar uma aplicação no Facebook, acesse o site https://developers.facebook.com. Registre uma aplicação (provavelmente com outro nome, pois o Meteor Bird acabei de registrar), inclua o domínio de redirect (pode ser http://localhost:3000) para que, após o usuário se autenticar, ele consiga voltar para tela principal do nosso sistema já autenticado. Ao registrar sua aplicação no Facebook, acesse o *dashboard* e procure pelos campos ID do aplicativo (`appId`) e o App Secret (`secret`). Copie e cole os respectivos valores da função anterior `ServiceConfiguration.configurations.insert()` dentro dos atributos `appId` e `secret`.

Após finalizar essa etapa, só nos resta testar. Reinicie o servidor e acesse a página da nossa aplicação. Repare que automaticamente uma nova opção de login (via **Facebook**) foi incluída pelo `Accounts UI`, então clique nesta opção e se autentique com sua conta do Facebook.

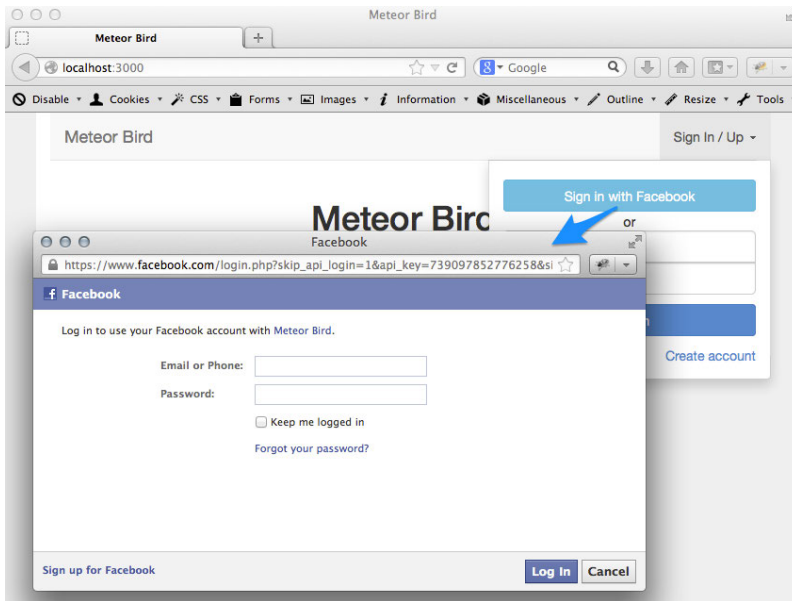



Figura 5.5: Autenticação via Facebook.

Parabéns! Fechamos mais um capítulo desenvolvendo um *signin* e *signup* em nossa aplicação, exploramos os principais recursos do objeto `Accounts` e seus packages auxiliares, também adicionamos uma autenticação via Facebook para agilizar o *signin* dos usuários. No próximo capítulo, criaremos um perfil de usuário e faremos alguns *refactorings* na autenticação do **Facebook** para consumir os dados dessa conta autocompletando nos dados do perfil.

Capítulo 6

Perfil do usuário

6.1 Criando template de perfil

Neste capítulo, criaremos uma tela para o perfil do usuário logado, que será exibido no bloco lateral esquerdo, constituído pelos campos: nome e descrição. De início, um novo usuário não terá esses dados completos. Para isso, vamos criar um bloco que apresenta o perfil (`Template.profileBox`), um template que será um formulário para atualizar o perfil (`Template.profileForm`) e, por último, um template que será o intermediador entre os templates anteriores (`Template.profile`). Começaremos criando o bloco do perfil, que será um subtemplate do diretório do template `home`, ou seja, vamos criar o diretório `client/views/home/profile`, dentro do qual implementaremos o HTML `client/views/home/profile/profile_box.html`. Este template vai utilizar a marcação `{{currentUser.profile}}` para apresentar

o perfil do usuário:

```
<template name="profileBox">
 {{#with currentUser.profile}}
  <h4>{{name}}</h4>
  <p>{{about}}</p>
  <button class="btn btn-block">
   Editar perfil
  </button>
 {{else}}
  <p>Perfil incompleto</p>
  <button class="btn btn-block">
   Completar perfil
  </button>
 {{/with}}
</template>
```

Assim, como criamos o bloco de exibição do perfil, também criaremos seu formulário para edição. Isso será no diretório `client/views/home/profile/profile_form.html`, seguindo a implementação:

```
<template name="profileForm">
 <form class="form-horizontal">
  <input type="text" class="form-control"
         placeholder="Nome">
  <input type="text" class="form-control"
         placeholder="Descrição">
  <button class="btn btn-success">
   Atualizar
  </button>
 </form>
</template>
```

Agora que temos os templates, precisamos criar o mediador entre eles. Ele basicamente utilizará a condicional da variável `Session.get('editProfile')` para mostrar o bloco do perfil quando seu valor for `false`, e mostrar o formulário quando seu valor

for `true`. Além disso, esta variável será visível pelos templates através de uma função *helper*. Para criar esta regra, primeiro crie o helper: `client/views/lib/helpers/profile.js` com a seguinte regra:

```
Template.profile.helpers({
 editProfile: function() {
  return Session.get("editProfile");
 }
});
```

Em seguida, crie o seu respectivo template em `client/views/home/profile/profile.html`:

```
<template name="profile">
 {{#if editProfile}}
  {{> profileForm}}
 {{else}}
  {{> profileBox}}
 {{/if}}
</template>
```

Finalmente, incluiremos perfil dentro de seu template pai, que é o `client/views/home/home.html`. Ele ficará na lateral esquerda através da tag `<aside>`, de acordo com código a seguir:

```
<template name="home">
 {{#if currentUser}}
  <aside class="col-sm-3">
   {{> profile}}
  </aside>
  <section class="col-sm-9">
   {{> post}}
   {{> timeline}}
  </section>
 {{else}}
  <div class="text-center">
   <h1>Meteor Bird</h1>
   <h3>Não esta logado? Clique em "Sign In/Up".</h3>
  </div>
```

```
  {{/if}}
</template>
```

Parabéns! Já temos os templates definidos sobre o perfil. Que tal agora dar vida aos templates? Por exemplo, em `profile_box.html` poderíamos dar funcionalidade aos links: "Editar perfil" e "Completar perfil". Eles serão responsáveis por apresentar o formulário e ambos executarão a mesma coisa, que será renderizar o `Template.profileForm` através do evento `"click button"` (clique do botão). Neste caso, estes links apenas irão atribuir o valor `true` para a função `Session.set("editProfile")` — e como `Session` é um objeto reativo, instantaneamente será feita essa transição entre o template de bloco do perfil para o template de formulário. Veja como fazer criando o código `client/lib/events/profile_box.js`:

```
Template.profileBox.events({
  "click button": function(e) {
    e.preventDefault();
    Session.set("editProfile", true);
  }
});
```

Com o formulário na tela, vamos agora codificar sua funcionalidade de atualizar os dados do perfil. Eles serão persistidos na collection `users` dentro do atributo `profile`, ou seja, em `users.profile`. Esta atualização será realizada via função `Meteor.user.update()`. Em seguida, mudaremos o estado de `editProfile` rodando a função `Session.set("editProfile", false)`, e também utilizaremos a função `e.preventDefault()` para cancelar a submissão padrão do formulário.

Criaremos a implementação dessas regras no código `client/lib/events/profile_form.js`, veja como fazer:

```
Template.profileForm.events({
  "submit form": function(e, template) {
    e.preventDefault();
    var inputs = template.findAll("input");
    Meteor.users.update(
```

```
    { _id: this.userId },
    { $set:
      { "profile.name": inputs[0].value,
        "profile.about": inputs[1].value }
    });
  }
  Session.set("editProfile", false);
});
```

Veja nas imagens como ficaram os layouts do perfil de usuário:

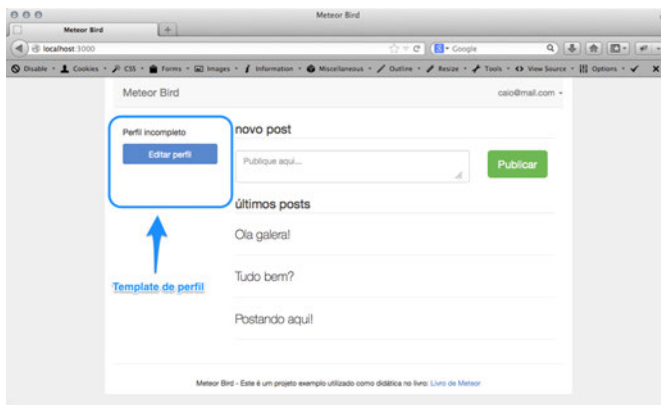



Figura 6.1: Tela de perfil incompleto.

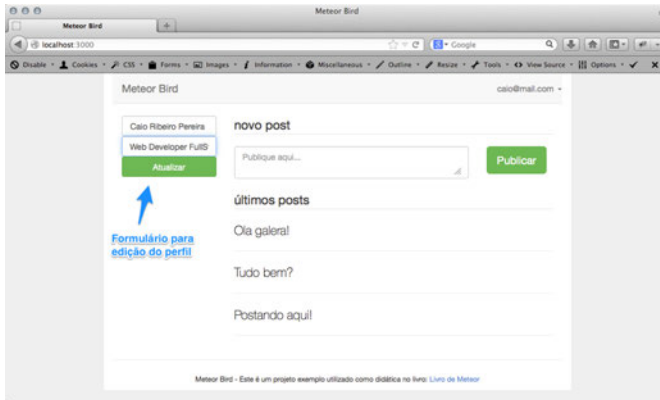



Figura 6.2: Atualizando dados do perfil.

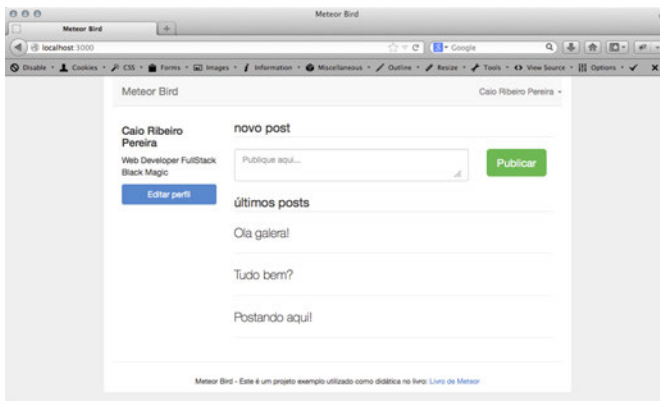



Figura 6.3: Perfil atualizado.

6.2 Autocompletando perfil via signin do Facebook

No capítulo anterior, foram criados dois modos de *signin* e *signup* na aplicação. O primeiro modo é o convencional cadastro de nome, e-mail e senha. Já o segundo, é o *signin* via conta do **Facebook**. Foi adicio-

nado o package `accounts-facebook` e também implementamos o arquivo `client/lib/config/accounts_auth.js`, definindo as permissões de consumo das informações de um usuário Facebook através da função `Accounts.ui.config()`. Nele, adicionamos a lista de permissões que permite consumir os dados de e-mail e dados público do usuário, tudo isso dentro do atributo `requestPermissions.facebook`.

Com base nessa implementação, iremos interceptar novamente o evento de cadastro de usuário, para autocompletar os dados vindos de uma conta do **Facebook**. Essa interceptação será apenas para identificar se o *signin* foi realizado via serviço **Facebook**; se sim, pegaremos o nome do usuário através do objeto `user.service.facebook.name`, caso contrário, retornamos o fluxo normal do *callback*. Para fazer isso, vamos extender o atual `server/lib/accounts.js` com o código a seguir:

```
Accounts.onCreateUser(function(options, user) {
 if (user.services.facebook) {
  var facebook = user.services.facebook;
  user['profile'] = {
   name: facebook.name
  };
 } else {
  user['profile'] = options.profile;
 }
 return user;
});
```

Feito isso, faça o teste: realize um login via Facebook com o qual, ao acessar a página principal, será exibido no perfil do usuário, o nome dele que veio de uma conta **Facebook**.

Foi mais uma feature implementada na rede social! Dessa vez nós exploramos um pouco a mais sobre manipulação de templates no *client-side*. Continue lendo! Nos próximos capítulos iremos trabalhar mais no *server-side* da aplicação, explorando alguns comandos principais do MongoDB, além de usar novos packages do Meteor.

Capítulo 7

Tela de perfil público do usuário

7.1 Adaptando rotas no projeto

Antes de começarmos a fundo a implementar a tela de perfil público do usuário, faremos alguns *refactorings* importantes para tornar o código da aplicação extensível e reutilizável, através do desenvolvimento de rotas.

Nesta etapa, faremos algumas alterações em códigos que já implementamos, mas antes adicionaremos um novo package, que será o mestre em gerenciar e manipular rotas. Seu nome é `iron-router`.

```
mrt add iron-router
```

Este é um framework que trata toda parte de rotas na aplicação. Mesmo que nossa aplicação seja em formato *single-page*, é possível torná-lo `multi-page` através de um roteamento de `paths`. Diferente das outras linguagens em que as rotas redirecionavam para páginas **HTML**, o

iron-router trabalha mais a fundo com manipulação de templates, ou seja, com ele, cada *redirect* será uma transição de um template para outro. Ele também permite ter um controle mais completo sobre uma rota, através da implementação do pattern classicamente conhecido por **controller**.

Voltando ao nosso projeto, começaremos a brincadeira criando o arquivo principal gerenciador de rotas, que se chama client/lib/routes.js. Veja a seguir como será seu código:

```
Router.map(function() {
 this.route('home', {
  path: '/'
  template: 'home',
  layoutTemplate: 'layout',
 });
});
```

Basicamente criamos a primeira rota da aplicação, que renderiza o template home ao acessar a rota raiz. Mas para que essa rota funcione corretamente, temos que adaptar a view principal para fazer funcionar o roteamento dos templates. Este roteamento inicia-se somente quando você criar o layoutTemplate incluindo nele a marcação {{> yield}}. No código anterior (client/lib/routes.js) foi definido quem será o layoutTemplate. Este template criaremos após a tag <body> na view principal; ou seja, abra o client/index.html e faça as seguintes modificações:

```
<head>
 <title>Meteor Bird</title>
</head>
<body>
</body>
<template name="layout">
 {{> header}}
 <main class="container">
  {{> yield}}
 </main>
 {{> footer}}
</template>
```

Se tudo der certo, a homepage da aplicação continuará sua exibição normalmente, mas agora a renderização dos templates serão controlados através das rotas no código `client/lib/routes.js`. Com essa pequena adaptação, deixamos a aplicação flexível para a inclusão de novas telas sem precisar adicionar códigos complexos no projeto. Essa flexibilidade ocorreu porque delegamos todo controle de templates para o `iron-router`.

Essa foi nossa primeira adaptação no projeto usando este framework. Continuando as adaptações para usar o `iron-router`, vamos migrar as funções de modelos existentes em alguns *helpers* para que elas sejam executadas na rota principal. Em `client/lib/routes.js` inclua o atributo `data` com a seguinte função:

```
Router.map(function() {
 this.route('home', {
  path: '/',
  template: 'home',
  layoutTemplate: 'layout',
  data: function() {
   return {
    posts: Post.list(Meteor.userId())
   }
  }
 });
});
```

Utilizando o atributo `data`, podemos rodar funções retornem objetos a serem renderizados nos templates. Agora que migramos essa função de listagem de posts, não há mais necessidade de manter seu antigo helper, então exclua o arquivo `client/lib/helpers/timeline.js`.

7.2 Perfil público do usuário

Agora que adaptamos nossa aplicação para o padrão de rotas, vamos focar na criação do perfil público do usuário. Para isso funcionar, cada usuário precisa ter uma rota própria e com identificador único para que tal perfil seja facilmente acessível por outros usuários.

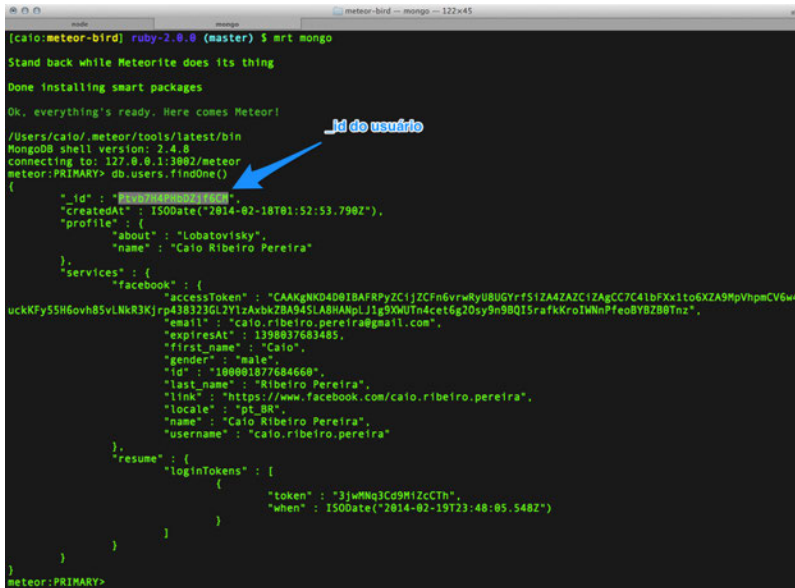
Depois dessa breve explicação, vamos fazer um *refactoring* com o objetivo de reutilizar a timeline atual dentro do template de perfil público. Para isso, vamos apenas mover o template `client/views/home/timeline.html` para o diretório acima que será em `client/views/timeline.html`.

Feita essa pequena alteração, vamos começar o desenvolvimento do perfil criando seu template. Crie o template de perfil em `client/views/user/user.html`:

```
<template name="user">
 {{#if user}}
  <div class="text-center">
   {{#with user.profile}}
    <h1>{{name}}</h1>
    <p>{{about}}</p>
   {{/with}}
   {{> timeline}}
  </div>
 {{else}}
  <h1>Usuário não identificado.</h1>
  <a href="{{pathFor 'home'}}">Homepage</a>
 {{/if}}
</template>
```

Nesta view, utilizamos uma nova marcação herdada do `iron-router`, que se chama `{{pathFor}}`. Ela basicamente retorna a string de uma url definida em sua respectiva rota. Neste caso retornou a url da `'home'` via função `{{pathFor 'home'}}`.

Agora vamos dar vida ao perfil público implementando suas operações necessárias para renderizar dados corretamente. Para isso, edite o `client/lib/routes.js`:

```
Router.map(function() {
 this.route('home', {
  path: '/',
  template: 'home',
  layoutTemplate: 'layout',
  data: function() {
   return {
```

```
    posts: Post.list(Meteor.userId())
   }
  }
 });
 this.route('user', {
  path: '/user/:_id',
  template: 'user',
  layoutTemplate: 'layout',
  data: function() {
    var _id = this.params._id;
    return {
      user: Meteor.users.findOne({_id: _id}),
      posts: Post.list(_id);
    }
   }
  });
});
```

Na rota `/user/:_id`, temos um pequeno detalhe que é o seu identificador: `":_id"`. Isto significa que a rota aceitará qualquer valor dentro do padrão desse path: `/user/[qualquer-valor]`. E esse valor será injetado na variável `this.params._id`. Esse pattern de criação de variáveis através de parâmetro nas rotas existe em diversos frameworks web como por exemplo: Rails, Express, VRaptor e outros.

Acabamos de criar o perfil. Para fazer um *test-drive*, cadastre um usuário no sistema ou entre utilizando uma conta no Facebook, e em seguida abra uma nova janela no terminal. Via comando `cd`, acesse o diretório do projeto para logo depois acessar o cliente MongoDB com o comando:

```
meteor mongo
```

Com o cliente rodando, execute a função `db.users.findOne()` para buscar o primeiro usuário da base, já com o resultado JSON formatado na tela. Do resultado, apenas copie o valor do campo `"_id"` e cole-o no browser formando o endereço http://localhost: 3000/user/[_id].

Figura 7.1: Pegando _id do usuário via CLI do MongoDB.

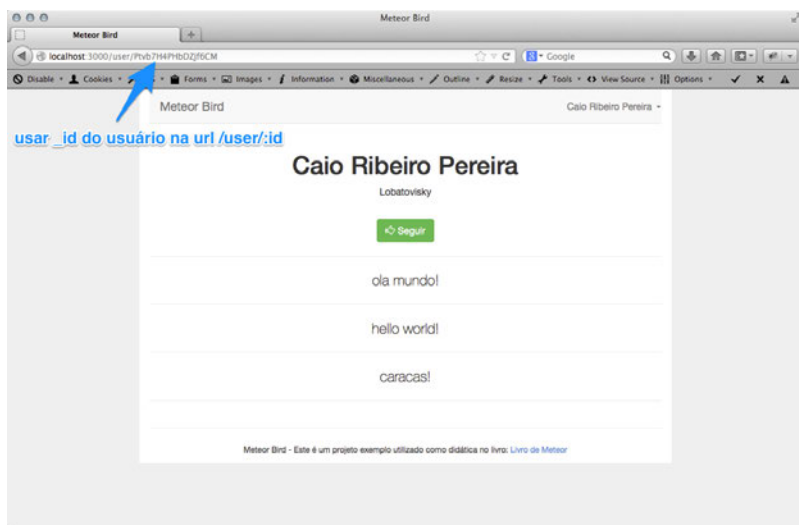



Figura 7.2: Inserindo _id na rota /users/:id.

Se tudo ocorrer bem, você verá a tela de perfil com os dados do usuário e seus últimos posts publicados.

E mais um capítulo finalizado, parabéns! Não pare de ler este livro, pois no próximo episódio iremos implementar a clássica funcionalidade de seguir e deixar de seguir um usuário.

Capítulo 8

Follow me I will follow you

8.1 Introdução sobre a funcionalidade

Este será um capítulo dedicado a explicar como criar a funcionalidade clássica de gerenciar seguidores na rede social. Afinal, o Meteor Bird é basicamente uma versão *light*, *diet*, sem açúcar do **Twitter** (http://twitter.com) , e caso você não saiba o significado de seguir um usuário, eis uma breve explicação: o objetivo principal de uma timeline é exibir seus posts e principalmente os posts de seus amigos; tais amigos são usuários que você **segue** na rede social. Por isso, seguir um usuário permite que você tenha acesso a conteúdos que ele publicar.

Com base nessa explicação, vamos neste capítulo focar apenas na construção dessa funcionalidade, que será dividida em pequenas funcionalidades listadas a seguir:

- Botão *follow* (seguir) e *unfollow* (deixar de seguir);
- Contador de seguidores no perfil do usuário.

Antes de começarmos a próxima seção, que tal preparar a timeline para receber o nome do usuário de cada post? Assim, visualizaremos facilmente o autor de cada post. Para isso, edite o `models/post.js` adicionando um novo parâmetro:

```
Post.publish = function(message, name) {
 var params = {
  message: message,
  time: new Date(),
  userId: Meteor.userId(),
  name: name
 };
 this.insert(params);
};
```

Em seguida, atualize o código de evento do post, o `client/lib/events/post.js`. Nele, pegaremos o nome do usuário logado utilizando a função `Meteor.user().profile.name`:

```
Template.post.events({
 "submit form": function(e, template) {
  e.preventDefault();
  var textarea = template.find("textarea");
  var name = Meteor.user().profile.name;
  Post.publish(textarea.value, name);
  textarea.value = "";
 }
});
```

E para finalizar, adicione uma tag no template da timeline para que exiba o nome do autor. É o que faremos editando a view `client/views/timeline.html`:

```
<template name="timeline">
 <div class="row">
```

```
  <fieldset class="col-sm-12">
   <legend>últimos posts</legend>
   {{#each posts}}
    <blockquote class="lead">
     <p>{{message}}</p>
     <small>Autor:
      <a href="/user/{{userId}}">{{name}}</a>
     </small>
    </blockquote>
    <hr>
   {{else}}
    <p class="lead text-center">
     Nenhum post publicado.
    </p>
   {{/each}}
  </fieldset>
 </div>
</template>
```

Pronto, veja como ficou a timeline:

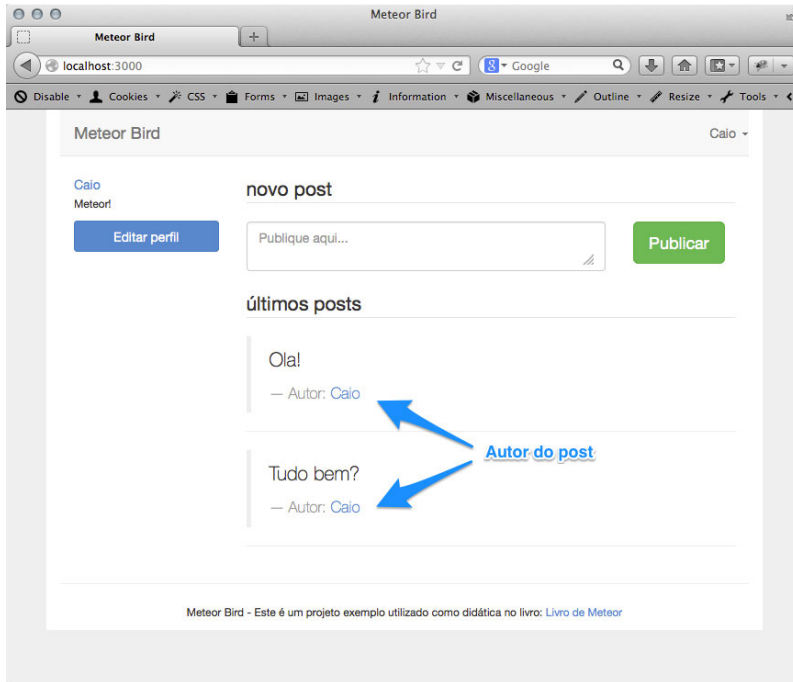



Figura 8.1: Exibindo autor do post na timeline.

8.2 Criando os botões de Follow e Unfollow

Depois de limpar toda a casa no capítulo anterior, já temos espaço suficiente para criar novas funcionalidades com orientação às rotas da aplicação. Nesta seção, vamos focar na implementação dos botões **Seguir** e **Deixar de seguir**.

Para começar, vamos codificar um novo modelo que chamaremos de `Friendship`. Ele terá uma estrutura simples, porém poderosa! Basicamente, ele vai armazenar apenas 2 campos:

- `userId`: `_id` do usuário logado;
- `friendId`: `_id` do usuário que se pretende seguir.

Com base nesses 2 campos, é possível criar as funcionalidades necessárias para seguir, deixar de seguir, contar quantos seguidores você tem e quantos

você segue. Mas indo por partes, vamos primeiro criar o modelo com as seguintes funções: `follow`, `unfollow` e `isFollowing`. Para fazer isso, crie este código em `models/friendship.js` seguindo o trecho:

```
Friendship = new Meteor.Collection('friendships');

Friendship.follow = function(friendId) {
 this.insert({
  userId: this.userId,
  friendId: friendId
 });
};
Friendship.unfollow = function(friendId) {
 this.remove({
  userId: this.userId,
  friendId: friendId
 });
};
Friendship.isFollowing = function(friendId) {
 return this.findOne({
  userId: this.userId,
  friendId: friendId
 });
};
```

Com base nessas 3 funções, atualizaremos a rota `/user` e criaremos 2 novas rotas para `follow` e `unfollow` de usuários. Abra e edite o `client/lib/routes.js` para fazer isso.

```
Router.map(function() {
 // código da rota "this.route('home')"
 this.route('user', {
  path: '/user/:_id',
  template: 'user',
  layoutTemplate: 'layout',
  data: function() {
   var _id = this.params._id;
   var isFollowing = Friendship.isFollowing(_id);
   Session.set('currentUserId', _id);
```

```
    Session.set('isFollowing', isFollowing);
    return {
      user: Meteor.users.findOne({_id: _id}),
      posts: Post.list(_id)
    }
   }
  });
  this.route('follow', {
   path: '/user/:_id/follow',
   action: function() {
    var _id = this.params._id;
    Friendship.follow(_id);
    this.redirect('/user/' + _id);
   }
  });
  this.route('unfollow', {
   path: '/user/:_id/unfollow',
   action: function() {
    var _id = this.params._id;
    Friendship.unfollow(_id);
    this.redirect('/user/' + _id);
   }
  });
});
```

Agora que temos as regras de `follow` e `unfollow`, vamos dar vida a eles criando seus respectivos botões no template de perfil público. Mas antes, falta criar algumas regras em helpers: uma regra é evitar que o usuário siga seu próprio perfil (através da função `canFollow()`), e a outra regra é a transição entre os botões **Seguir** e **Deixar de seguir** (via função `isFollowing()`). Vamos criá-las no novo helper `client/lib/helpers/follow_button.js`:

```
Template.followButton.helpers({
 canFollow: function() {
  var userId = Meteor.userId();
  return userId && Session.get('currentUserId') != userId;
 },
 isFollowing: function() {
```

```
  return Session.get('isFollowing');
 }
});
```

Esse helper será a base para implementar o template dos botões que criaremos em `client/views/user/follow_button.html` com os códigos a seguir:

```
<template name="followButton">
 {{#if canFollow}}
  {{#with user}}
   {{#if isFollowing}}
    <a href="{{pathFor 'unfollow'}}"
       class="btn btn-primary btn-large">
     Deixar de seguir
    </a>
   {{else}}
    <a href="{{pathFor 'follow'}}"
       class="btn btn-success btn-large">
     Seguir
    </a>
   {{/if}}
  {{/with}}
 {{/if}}
</template>
```

Repare nos links dos botões. Novamente utilizamos o `{{pathFor}}` e, incrivelmente, ele detectou o `_id` correto do path. Mas como ele fez essa mágica? O `iron-router` segue convenções de URLs REST, ou seja, quando criamos as rotas `/user/:_id/follow` e `/user/:_id/unfollow`, por convenção REST, estamos dizendo no path que o `:_id` é do `user`. Com base nisso, as funções: `{{pathFor 'follow'}}` e `{{pathFor 'unfollow'}}` incluíram o `_id` corretamente porque utilizamos a marcação `{{#with user}}` em volta desses links. Isso fez com que automaticamente o `_id` de um `user` apareça nos links.

Para mais detalhes sobre REST, recomendo a leitura desse livro:

http://www.casadocodigo.com.br/products/livro-rest

Para finalizar, adicione o template dos botões dentro de `client/views/user/user.html`:

```
<template name="user">
 {{#if user}}
  <div class="text-center">
   {{#with user.profile}}
    <h1>{{name}}</h1>
    <p>{{about}}</p>
   {{/with}}
   {{> followButton}}
   {{> timeline}}
  </div>
 {{else}}
  <h1>Usuário não identificado.</h1>
  <a href="{{pathFor 'home'}}">Homepage</a>
 {{/if}}
</template>
```

Que tal testarmos a aplicação seguindo alguns usuários? Vamos lá! Crie um novo usuário com o nome que você quiser. No meu caso, criarei o João e a Maria. Capture seus respectivos `_id` — pode ser manualmente, buscando pelo MongoDB através dos comandos:

```
meteor mongo
db.users.find().pretty()
```

O importante é que você acesse no browser o perfil público desses dois usuários, mas logado pela sua conta atual. Veja a seguir os perfis de João e Maria:

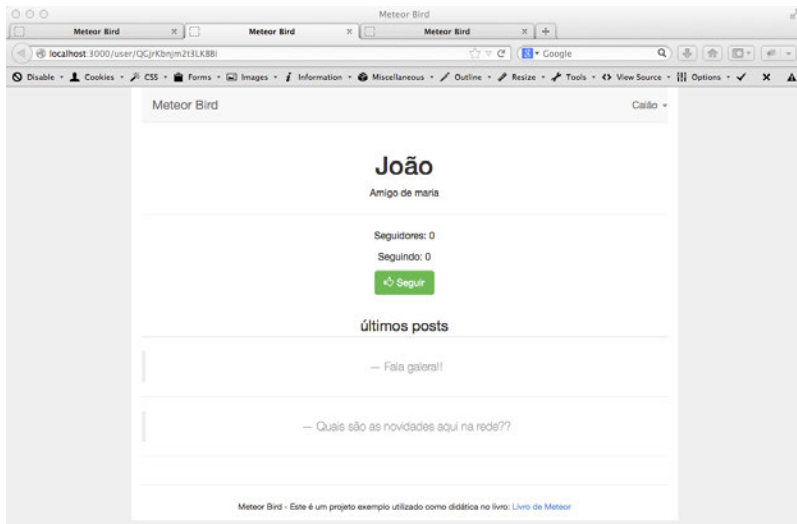



Figura 8.2: Perfil público de João.

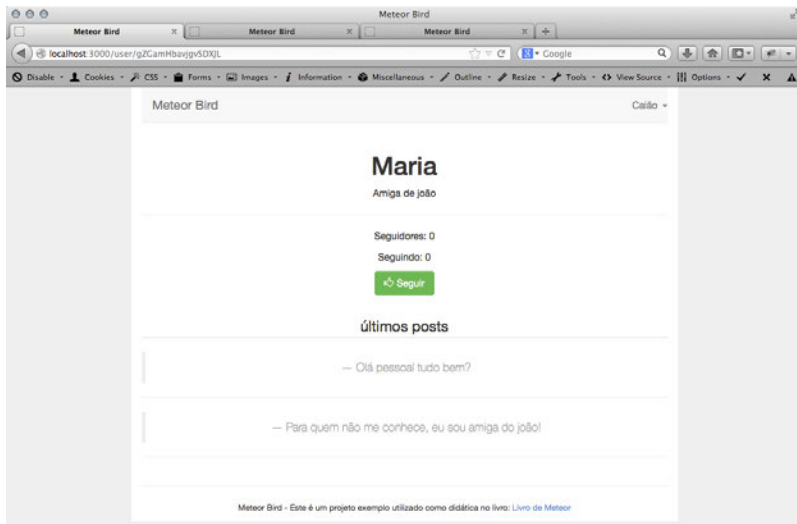



Figura 8.3: Perfil público de Maria.

Agora clique no botão **Seguir** dos dois perfis. Funcionou não é!? Legal! Mas depois de um certo tempo, você parou de falar com João e agora quer deixar de seguir seu perfil; nisso, você clicou no botão **Deixar de seguir** do perfil dele...

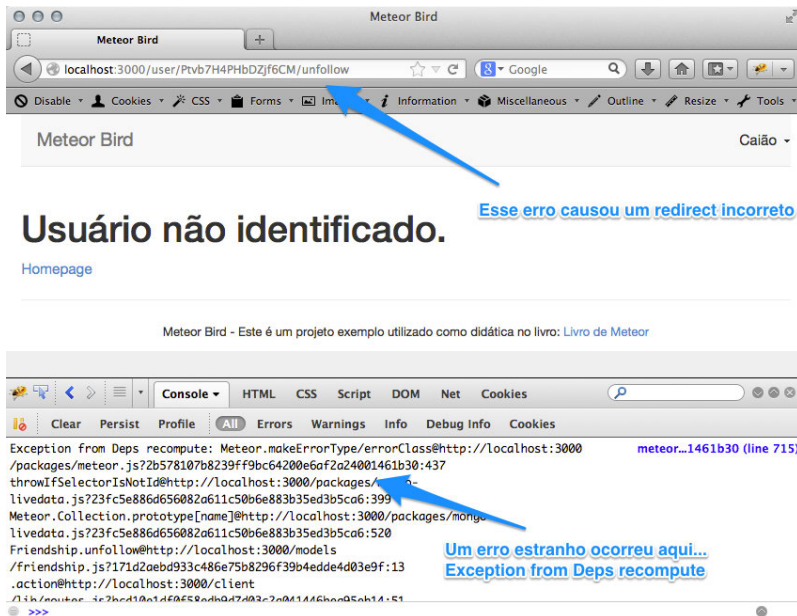



Figura 8.4: Erro: Exception from Deps recompute.

O que significa o erro **Exception from Deps recompute**? Esse erro ocorrerá sempre que surgir algum conflito no PubSub da aplicação. No momento, estamos usando o package default: `autopublish`. Ele cria automaticamente um `Publish` e um `Subscribe` para cada collection que usarmos no client-side da aplicação. Usá-lo em ambiente de produção é uma falha gravíssima de segurança pois vai liberar acesso a qualquer função do MongoDB via console de um browser. Utilize-o apenas para prototipar sua aplicação e, principalmente, para ter noção de quais operações serão utilizadas do banco de dados. Após definir todas as funções de banco de dados, remova este package e crie funções de publish e subscribe para cada operação utilizada das collections. Fique tranquilo, teremos um capítulo dedicado a como usar o PubSub e tam-

bém como preparar uma aplicação Meteor para ambiente de produção.

Voltando ao nosso problema atual, basicamente esse erro ocorre quando alguma conexão PubSub está quebrada, porém, como descobrir o que causou esse erro? Ao atualizar a tela atual do erro, surgiu um novo erro, esse sim é mais específico e causador desse caos na aplicação. Veja a imagem a seguir:

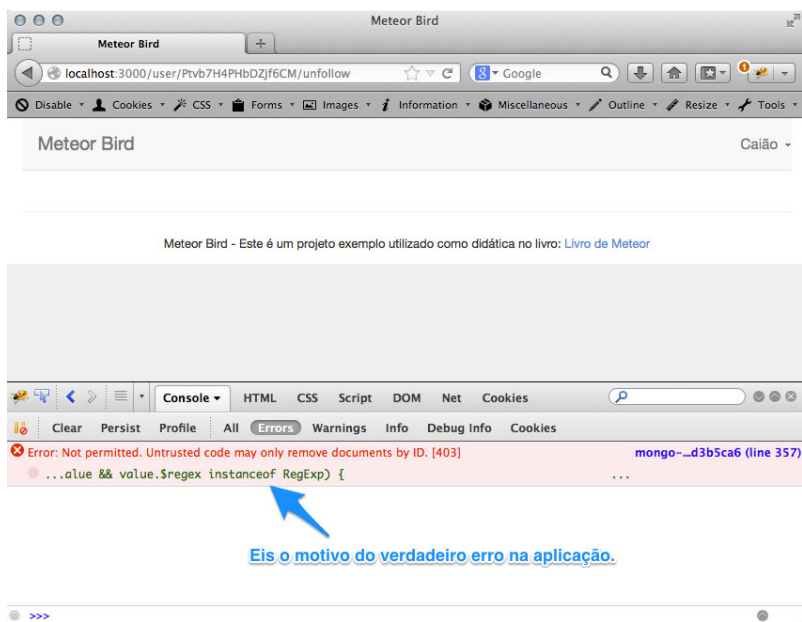



Figura 8.5: Erro: Not permitted. Untrusted code may only remove documents by ID.

Esse erro é um alerta padrão que o próprio Meteor gera quando se tenta atualizar ou remover dados de uma collection via client-side. É basicamente uma trava de segurança, que exige que operações desse tipo sejam executadas no server-side da aplicação para evitar que elas sejam acessíveis e adulteradas pelo cliente. Para entender melhor, recomendo que leia esse link da documentação oficial do Meteor:

http://docs.meteor.com/#allow

Agora que sabemos qual o real erro da aplicação, vamos focar em sua solução. Neste caso, vamos aprender a fundo como utilizar as funções

`Meteor.methods` e `Meteor.call`. Usamos o `Meteor.methods` para definir funções no servidor que serão chamadas pelo cliente através do `Meteor.call`. Fazendo uma comparação com outras tecnologias, podemos compará-lo com o uso do AJAX (*Asynchronous Javascript and XML*), em que o usamos no cliente para realizar requisições assíncronas no servidor. Ambos funcionam da mesma forma, a diferença é que a dupla `Meteor.methods` e `Meteor.call` possui um nível de abstração de alto nível, a ponto de apenas chamar a função do servidor e receber seu respectivo resultado.

Com base nesses conceitos, vamos resolver o problema que não permite executar a operação de *unfollow* de um usuário, afinal esta função foi programada para remover um registro do banco de dados. Na prática, vamos apenas encapsular a função `Friendship.unfollow` dentro de `Meteor.methods`, e migrar as demais funções de *insert* e *update* de outros modelos para este padrão também.

Crie o arquivo `server/methods.js` com o código a seguir:

```
Meteor.methods({
 followUser: function(friendId) {
  Friendship.follow(friendId);
 },
 unfollowUser: function(friendId) {
  Friendship.unfollow(friendId);
 },
 profileUpdate: function(name, about) {
  Meteor.users.update(
   {_id: this.userId},
   {$set: {
     "profile.name": name,
     "profile.about": about
   }}
  );
 },
 publishPost: function(message, name) {
  Post.publish(message, name);
 }
});
```

Agora que migramos as principais funções de inserir, atualizar e remover

dados, vamos mudar suas respectivas chamadas ao modelo para usarem o `Meteor.call`, de modo que não precisaremos mais usar o *package default* chamado `insecure`. Aliás, **jamais o utilize em ambiente de produção**, afinal ele permite a execução direta de funções `update` e `remove` das collections via browser. O `insecure` e `autopublish` (este último explicarei em detalhes no próximo capítulo) são packages pré-instalados por um único motivo, que é auxiliar a prototipagem rápida da aplicação. Não precisaremos mais dele então rode o seguinte comando:

```
mrt remove insecure
```

Começaremos o *refactoring* editando o `client/lib/routes.js`:

```
Router.map(function() {
 // código da rota "this.route('home')"
 // código da rota "this.route('user')"
 this.route('follow', {
  path: '/user/:_id/follow',
  action: function() {
   var _id = this.params._id;
   Meteor.call("followUser", _id);
   this.redirect('/user/' + _id);
  }
 });
 this.route('unfollow', {
  path: '/user/:_id/unfollow',
  action: function() {
   var _id = this.params._id;
   Meteor.call("unfollowUser", _id);
   this.redirect('/user/' + _id);
  }
 });
});
```

Em seguida, atualizaremos o código `client/lib/events/post.js`:

```
Template.post.events({
 "submit form": function(e, template) {
  e.preventDefault();
```

```
  var textarea = template.find("textarea");
  var name = Meteor.user().profile.name;
  Meteor.call("publishPost", textarea.value, name);
  textarea.value = "";
 }
});
```

E por último, edite o arquivo `client/lib/events/profile_form.js`:

```
Template.profileForm.events({
 "submit form": function(e, template) {
  e.preventDefault();
  var inputs = template.findAll("input");
  var name = inputs[0].value;
  var about = inputs[1].value;
  Meteor.call("profileUpdate", name, about);
  Session.set("editProfile", false);
 }
});
```

Pronto! Depois dessas dicas, agora você pode deixar de seguir o perfil do João.

8.3 Contador de seguidores no perfil

Para deixar o perfil do usuário mais rico em informações, vamos criar um template para informar o total de seguidores e quantos usuários ele segue. Para manter o template reusável, vamos criá-lo de forma que seja apresentado na tela do usuário logado e também no seu perfil público.

Crie o template `client/views/friendship.html` com o código a seguir:

```
<template name="friendship">
 <p>Seguidores: {{followers}}</p>
 <p>Seguindo: {{followings}}</p>
</template>
```

Adicionaremos esse template dentro do perfil público em `client/views/user/user.html`, através do marcador `{{> friendship}}`:

```
<template name="user">
 {{#if user}}
  <div class="text-center">
   {{#with user.profile}}
    <h1>{{name}}</h1>
    <p>{{about}}</p>
   {{/with}}
   {{> friendship}}
   {{> followButton}}
   {{> timeline}}
  </div>
 {{else}}
  <h1>Usuário não identificado.</h1>
  <a href="{{pathFor 'home'}}">Homepage</a>
 {{/if}}
</template>
```

E também adicionaremos este template em `client/views/home/home.html`:

```
<template name="home">
 {{#if currentUser}}
  <aside class="col-sm-3">
   {{> profile}}
   {{> friendship}}
  </aside>
  <section class="col-sm-9">
   {{> post}}
   {{> timeline}}
  </section>
 {{else}}
  <div class="text-center">
   <h1>Meteor Bird</h1>
   <h3>Não esta logado? Clique em "Sign In/Up".</h3>
  </div>
 {{/if}}
</template>
```

Feitas essas inclusões, vamos criar as funções para calcular o total de

`followers` e `followings`, através da collection `Friendship`. Abra e edite o código `models/friendship.js`:

```
Friendship = new Meteor.Collection('friendships');
// código da função "Friendship.follow()"
// código da função "Friendship.unfollow()"
// código da função "Friendship.isFollowing()"
Friendship.followings = function(userId) {
 return this.find({userId: userId}).count();
};
Friendship.followers = function(friendId) {
 return this.find({friendId: friendId}).count();
};
```

Por fim, vamos usá-las nas rotas `user` e `home`. Abra e edite o `client/lib/routes.js` e faça essa mágica da seguinte maneira:

```
Router.map(function() {
 this.route('home', {
  path: '/',
  template: 'home',
  layoutTemplate: 'layout',
  data: function() {
   var _id = Meteor.userId();
   return {
     posts: Post.list(_id),
     followers: Friendship.followers(_id),
     followings: Friendship.followings(_id)
   }
  }
 });
 this.route('user', {
  path: '/user/:_id',
  template: 'user',
  layoutTemplate: 'layout',
  data: function() {
   var _id = this.params._id;
   var isFollowing = Friendship.isFollowing(_id);
   Session.set('currentUserId', _id);
```

```
    Session.set('isFollowing', isFollowing);
    return {
      user: Meteor.users.findOne({_id: _id}),
      posts: Post.list(_id),
      followers: Friendship.followers(_id),
      followings: Friendship.followings(_id)
    }
  }
});
// código da rota "this.route('follow')"
// código da rota "this.route('unfollow')"
```

Veja como ficou bonito esse template:

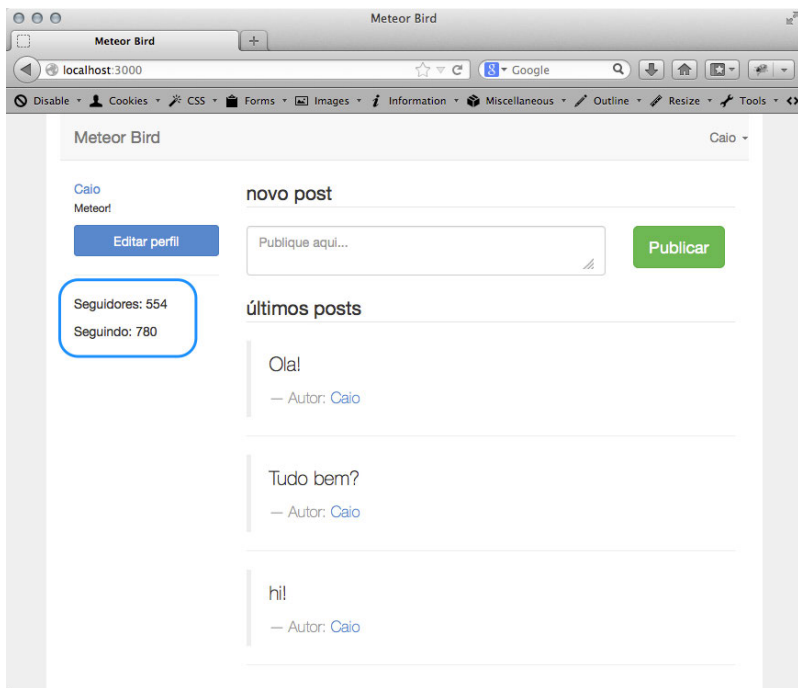



Figura 8.6: Contador de seguindo e seguidores.

8.4 Visualizando post de quem você seguir

Para finalizar o capítulo, vamos melhorar nossa timeline. Já temos todos os recursos implementados para seguir usuários, mas nada disso tem graça se não for possível visualizar os posts desses usuários — o mesmo vale para as pessoas que o estão seguindo nesta rede social por apenas um motivo: ver seus posts.

Na collection `Friendship` são mantidos todos os ids dos usuários que estão seguindo e sendo seguidos por outros usuários da rede. Para incrementar os posts da timeline de um usuário, basta apenas buscar todos os ids que o usuário estiver seguindo e, em seguida, usá-la como condição na consulta de posts, através da função `Post.list`. Dessa maneira, os resultados são ordenados em ordem cronológica decrescente, ou seja, do último para o primeiro post publicado.

Que tal implementar essa funcionalidade? Abra o arquivo `models/friendship.js`. Nele, vamos criar uma função que precisa retornar os ids de todos os perfis que o usuário principal estiver seguindo e incluir também o id do usuário principal. Para essa tarefa, crie a função `Friendship.timelineIds()`, que basicamente filtrará todos os documentos pertencentes ao `userId` em parâmetro. Porém, por questão de performance não pegaremos todos os dados de cada documento, e sim, somente o atributo `friendId`. Use a função `map` para transformar o array de documentos em um array de `friendId` e, por último, inclua o próprio `userId` neste mesmo array. Veja como fica a lógica desse código:

```
Friendship.timelineIds = function(userId) {
 var timelineIds = this.find({
  userId: userId
 }).map(function(f) {
  return f.friendId;
 });
 timelineIds.push(userId);
 return timelineIds;
};
```

Agora que temos essa função que retorna um array de ids de usuários, vamos usá-la para buscar os recentes posts desses usuários, tornando a time-

line funcional de verdade. Edite o `models/post.js` e modifique a função `Post.list` para retornar posts com base no array de ids de usuários em parâmetro (`userIds`), ordenando do último para o primeiro post publicado, e também por nome em ordem alfabética (através do atributo `{sort: {time: -1, name: 1}}`), veja abaixo:

```
Post.list = function(userIds) {
 return this.find(
  {userId: {"$in": userIds}},
  {sort: {time: -1, name: 1}}
 );
};
```

Para finalizar, atualize a listagem de posts nas rotas `'home'` e `'user'`. Abra o `client/lib/routes.js` e implemente o código a seguir:

```
Router.map(function() {
 this.route('home', {
  path: '/',
  template: 'home',
  layoutTemplate: 'layout',
  data: function() {
   var _id = Meteor.userId();
   var timelineIds = Friendship.timelineIds(_id);
   return {
     posts: Post.list(timelineIds),
     followers: Friendship.followers(_id),
     followings: Friendship.followings(_id)
   }
  }
 });
 this.route('user', {
  path: '/user/:_id',
  template: 'user',
  layoutTemplate: 'layout',
  data: function() {
   var _id = this.params._id;
   var timelineIds = Friendship.timelineIds(_id);
   var isFollowing = Friendship.isFollowing(_id);
```

```
    Session.set('currentUserId', _id);
    Session.set('isFollowing', isFollowing);
    return {
      user: Meteor.users.findOne({_id: _id}),
      posts: Post.list(timelineIds),
      followers: Friendship.followers(_id),
      followings: Friendship.followings(_id)
    }
  }
});
// código da rota "this.route('follow')"
// código da rota "this.route('unfollow')"
```

Com esse código implementado corretamente, finalizamos a listagem de posts da timeline. Ou seja, teremos um resultado semelhante a esta imagem:

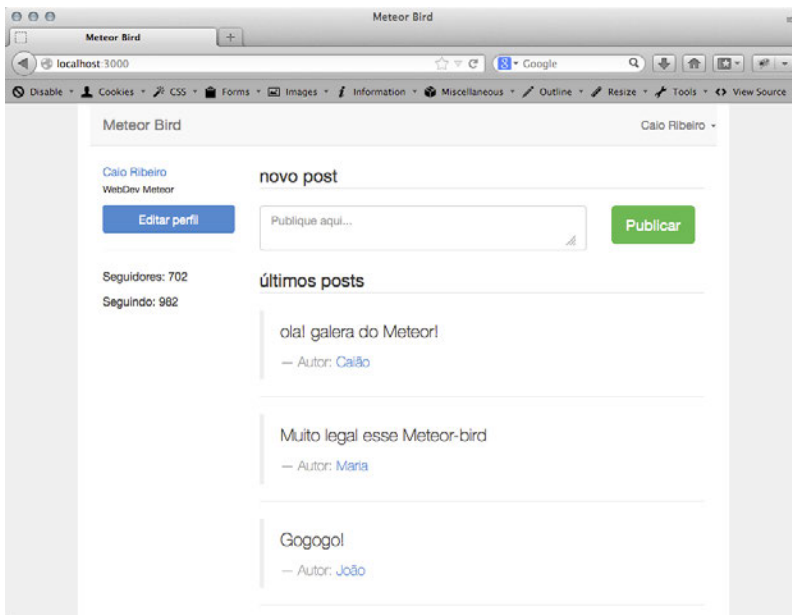



Figura 8.7: Timeline com posts multiusuários.

Parabéns! Você acabou de implementar mais uma *feature* com sucesso.

No próximo capítulo exploraremos a fundo a adoção de PubSub do Meteor para tornar nossa aplicação *ready for production!*, não perca!

Capítulo 9

Publications e Subscriptions

9.1 O que é PubSub?

O PubSub (*Publications and Subscriptions*), para quem não conhece, é um pattern cujo conceito é realizar mensageria através de dois personagens: um *publisher* (o publicador) e um *subscriber* (o assinante).

Esse conceito é utilizado em vários tipos de aplicações, como por exemplo: newsletters, chats e leitores de RSS. Ele funciona da seguinte maneira: o *publisher* é o responsável por emitir dados para todos os seus *subscribers*, e também um *subscriber* pode enviar mensagens para o *publisher*, fazendo com que o *publisher* trabalhe de forma mais específica com esse *subscriber*. Por exemplo, em um sistema newsletter, você, usuário, é o subscriber e, para receber atualizações desse serviço (neste caso receber atualizações de um publisher), você precisa informar seu e-mail para que ele saiba quem você é — ou seja, enviar alguns dados para o publisher, para que ele passe a enviar atu-

alizações para seu e-mail.

O Meteor já vem com uma biblioteca nativa de PubSub, então implementá-lo é bem tranquilo, além do mais, é uma obrigação fazer isso para que seja viável sincronizar dados entre cliente e servidor em um ambiente de produção.

Trabalhar corretamente com PubSub no Meteor vai garantir o desenvolvimento de uma aplicação sustentável, visto que uma implementação mal-feita vai tornar o sistema uma carroça, de tão lenta que será a velocidade na sincronização dos dados. Outro detalhe de extrema importância é a segurança dos dados, pois uma publicação que liberar dados errados para o usuário vai gerar vazamento de informações na aplicação. Por isso, peço que leia este capítulo com atenção, para aprender como funciona esse mecanismo do Meteor.

9.2 Entendendo seu mecanismo

Imagine um cenário em que nossa aplicação seja acessada por mais de 10 mil usuários. Nele, temos um usuário que segue muitos usuários populares, cerca de mil. Como sabemos, o objetivo do PubSub é sincronizar dados entre o servidor MongoDB e o cliente Minimongo, o que faz com que o Meteor trafegue apenas dados seguindo seu princípio **data on-the wire**. Essa sincronização se faz através do protocolo DDP (*Data Distribution Protocol*). Em resumo, este protocolo é o que faz a biblioteca PubSub enviar e receber dados através de objeto JSON em tempo real.

Um pouco sobre DDP

Uma aplicação Meteor em execução trabalha simultaneamente com dois tipos de servidores:

- **Servidor HTTP**: responsável por servir conteúdo estático e tratar requisição do cliente. Internamente, ele utiliza o framework Connect (http://www.senchalabs.org/connect) do Node.js para realizar esse trabalho.
- **Servidor DDP**: este é um servidor inteiramente dedicado a lidar com PubSub, Meteor.methods e funções do MongoDB. Seu mecanismo funciona graças ao SockJS (http://sockjs.org) , que trabalha transportando os dados entre o cliente e o servidor em tempo real.

Voltando ao cenário em que temos uma rede social com 10 mil usuários, queremos mostrar na timeline os últimos posts de cada usuário seguido, e para isso, o mais óbvio é sincronizar os 10 mil usuários entre cliente e servidor, isso é, para cada usuário logado no sistema. Ao fazer essa prática, você terá um gargalo muito grande na aplicação, visto que a cada usuário teremos que carregar no cliente os dados de 10 mil usuários. Esse é o comportamento padrão do `autopublish` habilitado. Afinal, somente quem desenvolve a aplicação conseguirá filtrar os dados corretamente.

O ideal, neste caso, é publicar apenas o montante de usuários coerente ao total de usuários seguidos, fazendo com que cada um sincronize de forma eficiente somente os dados necessários para ele.

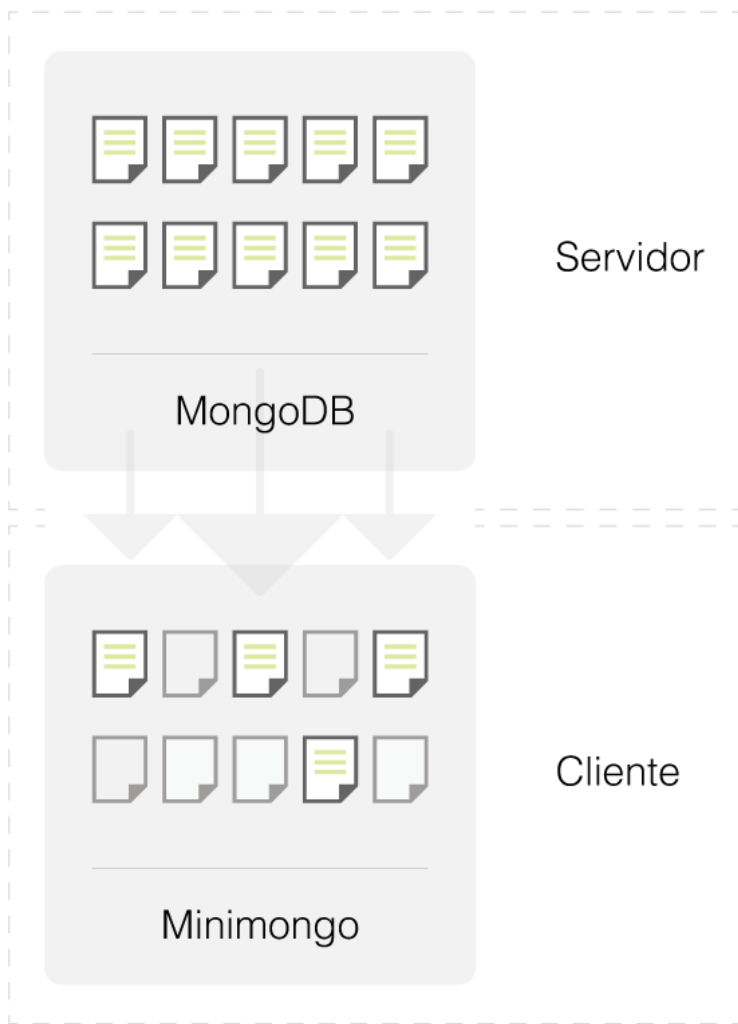



Figura 9.1: Publicando somente o necessário para o cliente.

Outro detalhe é que vale a pena sincronizar um montante aceitável de dados, visto que toda manipulação realizada no *client-side* será executada de

forma rápida graças aos mecanismos de *prefetching* e *model simulation* do Minimongo, que basicamente faz um *cache* do resultado de uma publicação, mantendo no cliente esses os dados para uso e manipulação.

9.3 Adaptando o PubSub no projeto

Para entender na prática os conceitos de PubSub, realizaremos algumas modificações no Meteor Bird.

A primeira tarefa será remover o package `autopublish`, afinal ele deve ser usado apenas para prototipar um sistema, e como já terminamos a prototipação deste projeto, não há motivos de usá-lo mais. Aliás **deixá-lo instalado em ambiente de produção vai gerar uma grande brecha de segurança**, tornando possível **rodar qualquer consulta do banco de dados através do console JavaScript de um browser**, ou seja, não há mais necessidade de manter essa dependência na aplicação, pois não queremos que qualquer um faça uma festa indesejada, adulterando os registros do banco de dados ou pior excluir todos os registros conforme a imagem abaixo:

Figura 9.2: Cuidado com autopublish! Desinstale-o em produção.

Desinstale o `autopublish`:

```
mrt remove autopublish
```

Reinicie o servidor Meteor. Repare que após remover o `autopublish` a aplicação vai parar de funcionar corretamente.

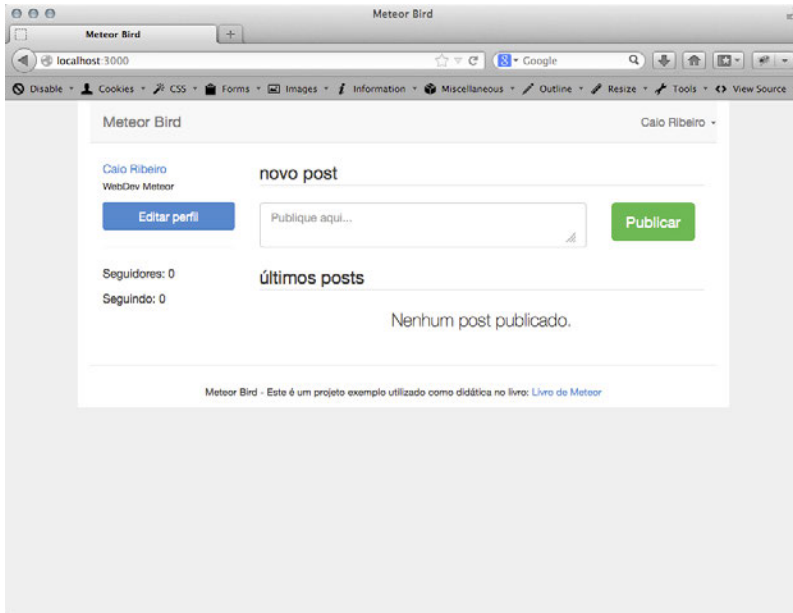



Figura 9.3: Ué! Por que a timeline parou de funcionar?.

Fique calmo! Este problema ocorreu porque nenhum PubSub foi criado. Antes, com `autopublish` habilitado, tudo era gerado automaticamente, e agora teremos que criar um PubSub para cada consulta no banco de dados.

Primeiro, criaremos os publishers, pois estes são os principais e mais fáceis de implementar no *server-side* do sistema. Crie o código `server/publications.js`, que vai conter todos publishers da aplicação. Para tornar as publicações reativas, vamos englobar cada função `Meteor.publish` no callback da função `Meteor.autorun`, fazendo com que a qualquer alteração nos modelos ocorra um *recomputation* de cada publicação.

```
Meteor.autorun(function() {

 Meteor.publish("posts", function(_id) {
  var timelineIds = Friendship.timelineIds(_id);
  return Post.list(timelineIds);
```

```
 });

 Meteor.publish("friendship", function(_id) {
  return Friendship.followersAndFollowings(_id);
 });

 Meteor.publish("isFollowing", function(_id) {
  return Friendship.isFollowing(_id);
 });

 Meteor.publish("user", function(_id) {
  return Meteor.users.find({_id: _id});
 });

});
```

Repare que em `Meteor.publish("friendship")` estamos utilizando uma nova função: `Friendship.followersAndFollowings`. Seu objetivo é retornar todos os `followers` e `followings` do usuário, pois será este resultado que publicaremos do servidor para o cliente assinar. Para implementá-la, edite o arquivo `models/friendship.js` incluindo a função a seguir:

```
Friendship.followersAndFollowings = function(_id) {
 return this.find({$or: [{userId: _id}, {friendId: _id}]});
};
```

Já no *client-side*, teremos que adaptar as funções de algumas rotas, e para isso utilizaremos uma nova função do `iron-router`, chamada de `onBeforeAction`. No callback desta função, incluiremos todos os `subscribes` necessários para as rotas `'home'` e `'user'`. Ao incluir os `subscribers` necessários, algumas funções de modelos serão excluídas da rota, visto que elas já serão publicadas pelo servidor — este é o caso da função `Friendship.timelineIds`. Edite o `client/lib/routes.js` e faça as seguintes alterações:

```
Router.map(function() {
 this.route('home', {
```

```
  path: '/',
  template: 'home',
  layoutTemplate: 'layout',
  onBeforeAction: function() {
   var _id = Meteor.userId();
   this.subscribe("posts", _id);
   this.subscribe("friendship", _id);
  },
  data: function() {
   var _id = Meteor.userId();
   return {
    posts: Post.find({}),
    followers: Friendship.followers(_id),
    followings: Friendship.followings(_id)
   }
  }
 });
 this.route('user', {
  path: '/user/:_id',
  template: 'user',
  layoutTemplate: 'layout',
  onBeforeAction: function() {
   var _id = this.params._id;
   this.subscribe("posts", _id);
   this.subscribe("friendship", _id);
   this.subscribe("isFollowing", _id);
   this.subscribe("user", _id);
  },
  data: function() {
   var _id = this.params._id;
   var isFollowing = Friendship.isFollowing(_id);
   Session.set('currentUserId', _id);
   Session.set('isFollowing', isFollowing);
   return {
    posts: Post.find({}),
    user: Meteor.users.findOne({_id: _id}),
    followers: Friendship.followers(_id),
    followings: Friendship.followings(_id)
   }
```

```
  }
 });
 // função this.route('follow');
 // função this.route('unfollow');
});
```

Além da exclusão de `Friendship.timelineIds`, tivemos uma outra alteração de funções nas rotas. Repare que a função `Post.list` foi substituída pela função `Post.find({})`. E o mais engraçado é que esta função não vai retornar os posts de todo mundo e sim os posts da função `Post.list` que foi implementado na publicação `Meteor.publish('posts')`. O motivo desse comportamento é que funções de collections no *client-side* sempre executarão com base nos resultados que receberem de uma publicação *server-side*. Isto é, já que criamos `Meteor.publish("post")` para retornar apenas os posts do usuário e de quem ele estiver seguindo, ao assinarem essa publicação, o modelo `Post` no *client-side* somente terá esses dados para manipular, então se for usar `Post.list()` ou `Post.find({})` os resultados serão o mesmo.

Praticamente terminamos a prototipagem de nosso projeto, o Meteor Bird. Nos próximos capítulos focaremos em trabalhar a aplicação para colocalá no ar com qualidade, ou seja, implementaremos testes e realizaremos alguns *tunings* para garantir que nossa aplicação funcione perfeitamente em um ambiente de produção.

Capítulo 10

Testes, testes e mais testes

10.1 Frameworks de testes para o Meteor

Neste capítulo vamos explorar um tema que é muito importante não apenas para o Meteor, mas para qualquer linguagem de programação: vamos falar sobre criação de testes.

O Meteor possui alguns framework de testes. Atualmente existem quatro em destaque: **RTD**, **TinyTest**, **Mocha Web** e **Laika**.

Esses frameworks permitem testar um mesmo trecho de código no cliente e no servidor, o que faz essas ferramentas se tornarem totalmente específicas para trabalhar com o Meteor.

Por que devemos criar teste de códigos

É muito importante que todo desenvolvimento de classes ou módulos seja acompanhado com seu respectivo código de testes. Testar um código faz com que você identifique e corrija diversos erros antes de colocar o novo código no ar. Criar testes é uma prática extremamente recomendada e adotada por muitas empresas e projetos open-source. Além de garantir que seu código funcionará como esperado, isso pode influenciar no código que você esta criando, fazendo com que você identifique e melhore o design do código. Outra vantagem dos testes é que tornam o projeto mais flexível e preparado para mudanças de regras de negócio, visto que os testes apontarão quais funções quebrarão ao mudar tal trecho de código na aplicação.

Existem vários livros explicando como e quando criar testes, além de ensinar técnicas para tornar um código menos acoplado e facilmente testável. Na editora Casa do Código existem três livros sobre TDD (*Test-Driven Development*):

- Test-Driven Development com Java

  http://www.casadocodigo.com.br/products/livro-tdd

- Test-Driven Development com .NET

  http://www.casadocodigo.com.br/products/livro-tdd-dotnet

- Test-Driven Development com Ruby

  http://www.casadocodigo.com.br/products/livro-tdd-ruby

Estes livros, por exemplo, ensinam boas práticas de testes um com a plataforma Java, um com a plataforma .NET e um com Ruby. Apesar de serem específicos para uma determinada linguagem, grande parte de seus conceitos são aplicáveis para qualquer linguagem de programação.

10.2 Primeiros passos com Laika

O Laika é um framework fácil de configurar e de se trabalhar. Ele foi inspirado no framework Mocha (http://visionmedia.github.io/mocha) do Node.js, ou seja, ele herdou algumas características como criar testes no estilo TDD (*Test-Driven Development*) ou BDD (*Behavior-Driven Development*), customização do *report* dos testes e criação de um arquivo de configuração.

Para instalar o Laika, **é necessário instalar o PhatomJS primeiro**, pois ele será muito utilizado na bateria de testes do *client-side* do projeto. Para usá-lo, recomendo que leia seu manual de instalação neste link (http://phantomjs.org/download.html) para instalá-lo de acordo com seu sistema operacional.

O Laika é um módulo Node.js, presente no site NPM. Instale-o em modo global para habilitar seu CLI (*Command Line interface*), para isso execute o comando:

```
npm install -g laika
```

O Laika roda em uma instância default do serviço MongoDB, ou seja, ele vai trabalhar com um banco de testes separado do banco atual da aplicação. Aliás, ele vai rodar em cima do MongoDB default instalado em sua máquina.

A dica a seguir **não é obrigatória, faça apenas se quiser e, principalmente, se achar necessário**. É possível configurar o MongoDB para rodar os testes mais rápido em ambiente de desenvolvimento. Para isso, finalize o serviço atual do MongoDB e inicie-o utilizando este comando:

```
mongod --smallfiles --noprealloc --nojournal
```

Com nosso framework de teste instalado, vamos criar os nossos primeiros testes. Focaremos apenas em implementar testes para os modelos, pois é lá que foram criadas as principais regras de negócio da aplicação.

10.3 Criando testes

Criar testes no Meteor é bem diferente do que em outras tecnologias como Ruby, Java, PHP ou Node.js. Isso ocorre porque, no Meteor, um modelo é compartilhado e acessado tanto no servidor como no cliente, e por isso,

dependendo das regras de negócio, será necessário criar testes para verificar como um modelo se comporta em ambos os lados da aplicação.

Testando o modelo Friendship

Começando nossa codificação, vamos criar os testes para o modelo `Friendship`, visto que ele possui muitas funções importantes que precisam ser testadas. Criaremos seu respectivo teste em `tests/friendship.js`, onde testaremos primeiro a função `Friendship.follow`. Para simular esse teste, teremos que utilizar um objeto `server` para que através do callback de `server.eval`, o modelo `Friendship` rode a função de observação de seu estado. Neste teste observaremos apenas se o modelo recebeu um novo registro, através da função `added` e caso essa função seja executada ela irá emitir um evento enviando o estado do modelo via função `emit("added", obj)`. Essa observação somente será executada quando no *client-side*, através do callback `client.eval`, a função `Meteor.call('followUser')` for executada. Quando o evento `emit("added")` for acionado, utilizaremos o callback de `server.once("added")` para realizar os devidos testes através das funções existentes do objeto `assert`. No teste abaixo apenas utilizaremos a função `assert.equal()` para verificar se o `friendId` e `userId` são do cliente que a executou.

```
var assert = require("assert");

suite("Friendship", function() {
 test("follow", function(done, server, client) {

  server.eval(function() {
   Friendship.find().observe({
    added: function(obj) {
     emit("added", obj);
    }
   });
  });

  server.once("added", function(obj) {
   assert.equal(obj.friendId, "123");
```

```
    assert.equal(obj.userId, this.userId);
    done();
   });

   client.eval(function() {
    Meteor.call("followUser", "123");
   });

  });
 });
```

Basicamente, este teste simulou um cliente rodando a função `Friendship.follow("123")` que está encapsulada dentro da função `Meteor.call("followUser", "123")`, afinal nesta etapa já removemos o package `insecure`, então qualquer função de inserção, atualização ou exclusão será executada no cliente via `Meteor.call`.

Implementar o teste da função `Friendship.unfollow` é um pouco parecido com a anterior. A diferença é que primeiro teremos que registrar um `follow` e, em seguida, fazer um `unfollow` do mesmo `userId`, para depois observar se no modelo `Friendship` houve uma exclusão desse registro. O código a seguir exemplifica na prática esse teste:

```
var assert = require("assert");

suite("Friendship", function() {
 // Código: test("follow")...

 test("unfollow", function(done, server, client) {
  server.eval(function() {
   Friendship.find().observe({
    removed: function(obj) {
     emit("removed", obj);
    }
   });
  });

  server.once("removed", function(obj) {
   assert.equal(obj.friendId, "A");
```

```
    assert.equal(obj.userId, this.userId);
    done();
  });

  client.eval(function() {
    Meteor.call("followUser", "A", function() {
      Meteor.call("unfollowUser", "A");
    });
  });

 });
});
```

E como faz para testar a função `Frienship.isFollowing()`? Essa é mais simples: faremos um `follow` com `userId` válido e vamos rodar a função `Frienship.isFollowing` duas vezes, uma com `userId` utilizado pelo `follow` e um `userId` inexistente. No final, checaremos se um objeto existe e se o outro não existe.

```
var assert = require("assert");

suite("Friendship", function() {
 // Código: test("follow")...
 // Código: test("unfollow")...

 test("isFollowing", function(done, server, client) {
  server.eval(function() {
   Friendship.find().observe({
    added: function(obj) {
     var obj1 = Friendship.isFollowing("123");
     var obj2 = Friendship.isFollowing("321");
     emit("check", obj1, obj2);
    }
   });
  });

  server.once("check", function(obj1, obj2) {
   assert.notEqual(obj1, undefined);
   assert.equal(obj2, undefined);
```

```
   done();
  });

  client.eval(function() {
   Meteor.call("followUser", "123");
  });
 });
});
```

Outro teste que vale a pena implementar é para a função `Friendship.timelineIds`. Sua regra é retornar todos os `friendIds` de um `userId` incluindo também o próprio `userId` no array. Para simular esse teste, o cliente terá que seguir dois usuários `"A"` e `"B"`. No servidor, deixaremos o modelo `Friendship` observando via função `Friendship.find().observe()` cada novo registro através do evento `addedAt`. Este é um evento mais preciso do que o `added`, pois em seu callback ele retorna o objeto cadastrado, seu índice de registro (`index`) e também um objeto anterior a ele (`before`). Com isso, faremos com que o servidor emita o evento `emit('timelineIds')` somente se o `index` for igual a 1, ou seja, somente se já existirem 2 registros de `Meteor.call("followUser")` na base, assim poderemos executar a função `Friendship.timelineIds` e realizar os testes em cima do resultado dessa função.

```
var assert = require("assert");

suite("Friendship", function() {
 // Código: test("follow")...
 // Código: test("unfollow")...
 // Código: test("isFollowing")...

 test("timelineIds", function(done, server, client) {
  server.eval(function() {
   Friendship.find().observe({
    addedAt: function(obj, index, before) {
     if (index == 1) {
      var ids = Friendship.timelineIds(this.userId);
      emit('timelineIds', ids);
```

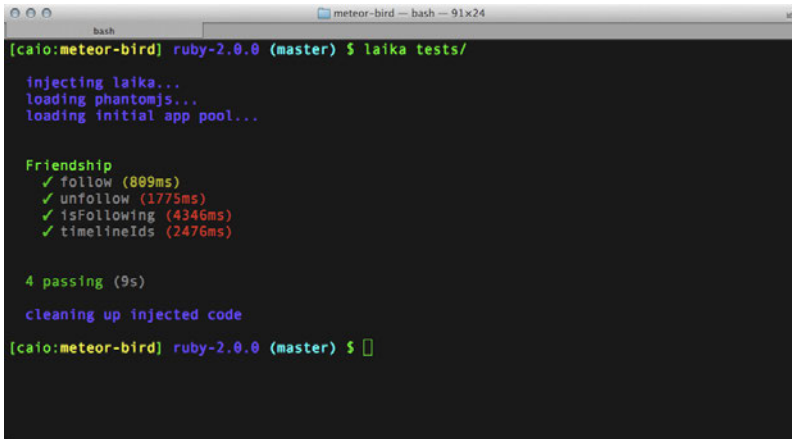
```
    }
   }
  });
 });

 server.once('timelineIds', function(ids) {
  assert.equal(ids.length, 3);
  assert.equal(ids[0], "A");
  assert.equal(ids[1], "B");
  assert.equal(ids[3], this.userId);
  done();
 });

 client.eval(function() {
  Meteor.call("followUser", "A", function() {
   Meteor.call("followUser", "B");
  });
 });
});
});
```

Ok! Já temos testes suficientes para o modelo `Friendship`, que tal rodá-los para ver o que acontece? Execute o comando a seguir:

```
laika tests
```

Se tudo der certo, você verá o resultado dos testes semelhante à imagem:

```
[caio:meteor-bird] ruby-2.0.0 (master) $ laika tests/

  injecting laika...
  loading phantomjs...
  loading initial app pool...


  Friendship
    ✓ follow (809ms)
    ✓ unfollow (1775ms)
    ✓ isFollowing (4346ms)
    ✓ timelineIds (2476ms)


  4 passing (9s)

  cleaning up injected code

[caio:meteor-bird] ruby-2.0.0 (master) $
```

Figura 10.1: Resultados dos testes no modelo Friendship.

10.4 Desafio: testar o modelo Post

Basicamente foram explicadas as funções necessárias do Laika e MongoDB suficientes para se criar testes no modelo `friendship.js`.

Como lição de casa, ofereço um desafio! Você terá que criar testes para o modelo `post.js`. Desenvolva os testes em cima das 2 funções: `Post.list` e `Post.publish`. Para isso, desenvolva os testes no arquivo em `tests/post.js` e mãos na massa!

Obs.: Se você estiver com problemas, participe do grupo deste livro enviando suas dúvidas através desse link:

https://groups.google.com/forum/#!forum/livro-meteor

Caso queira ver como fica o teste mínimo deste modelo, você pode visualizar o código-fonte desse teste através desse link:

https://github.com/caio-ribeiro-pereira/meteor-bird/blob/master/tests/post.js

Boa sorte no desafio e continue lendo, pois ainda tem muito caminho pela frente!

Capítulo 11

Integração contínua no Meteor

No capítulo anterior, aprendemos o quão importante é criar testes. Criarmos na prática testes unitários nas funções dos modelos da nossa aplicação, e tudo isso foi apresentado explorando as funcionalidades do framework: Laika.

Neste capítulo vamos incrementar nosso projeto, adicionando-o a um serviço de integração contínua, no qual deixaremos automática a execução dos testes unitários toda vez que enviarmos um novo *commit* para este projeto.

O objetivo deste capítulo é apenas explicar como configurar um ambiente de integração contínua utilizando o Meteor. Utilizaremos como exemplo o serviço Travis-CI (http://travis-ci.com) , porém as dicas também servem em outros serviços de integração contínua.

O Travis-CI é um serviço online gratuito apenas para repositórios públicos. Em casos de repositórios privados, é necessário desembolsar uma grana para tanto.

Como nosso projeto será apenas para fins educativos, vamos hospedá-lo em um repositório público, por exemplo, em um repositório público do Github (http://github.com) .

11.1 Rodando Meteor no Travis-CI

Configurar uma aplicação Meteor no Travis-CI requer alguns parâmetros adicionais no arquivo `.travis.yml`, mas não é nada complexo. A começar, você vai configurá-lo como `language: node_js` (Node.js como linguagem de programação), `service: mongodb` (MongoDB como serviço). Até aí tudo bem, não há segredo! Mas como é que o Travis-CI vai instalar o Meteor, Meteorite, Laika e no final rodar os testes?

Neste caso, será necessário incluir um script adicional dentro do campo `before_install`. Este script será baixado via `curl` e executado via `/bin/sh` (ambiente *bash script*). Ele vai baixar e instalar o Meteor, Meteorite, Laika e, para finalizar, executar todos os testes da aplicação.

JavaScript e Node.js no Travis-CI

Na documentação oficial do Travis-CI, existe um artigo explicando alguns tipos de configurações para Node.js. Entre eles, existe um modelo pronto para rodar um projeto Meteor executando testes via módulo Laika. Para conhecer outras configurações para ambiente JavaScript ou Node.js, veja este link:

http://docs.travis-ci.com/pt-BR/user/languages/javascript-with-nodejs

Configurando o Travis-CI

Em nosso projeto, faremos uma pequena alteração, e o resto serão apenas configurações no serviços Travis-CI e Github (O Travis-CI já possui signin direto de uma conta Github). No diretório raiz do projeto, crie o arquivo `.travis.yml` contendo o seguinte código:

```
language: node_js
```

```
node_js:
 - "0.10"
before_install:
 - "curl -L http://git.io/3l-rRA | /bin/sh"
services:
 - mongodb
```

Obs.: a URL http://git.io/3l-rRA utilizada para preparar o ambiente é apenas uma URL encurtada que direciona-se para o endereço:

https://raw.github.com/arunoda/travis-ci-laika/master/configure.sh

E o arquivo `configure.sh`, internamente, visa a instalar o Meteor, Meteorite e Laika no Travis-CI, ou seja, em código, este script basicamente executa os seguintes comandos:

```
#!/bin/sh

#configuring the system
wget
  https://raw.github.com/arunoda/travis-ci-laika/master/Makefile

#install meteor
curl https://install.meteor.com | /bin/sh

#installing meteorite
npm install -g meteorite

#installing laika
npm install -g laika
```

Agora nos resta cadastrar uma conta no Github e, em seguida, no Travis-CI. **Você também pode utilizar uma conta existente.**

Atenção: não entrarei em detalhes técnicos, visto que esses serviços com o passar do tempo podem mudar seus meios de interatividades e também suas interfaces visuais.

Mas basicamente, você terá que fazer as seguintes tarefas:

1) Criar um repositório no Github;

2) Fazer um *commit* do código-fonte do Meteor Bird para o repositório Github;

3) Associar sua conta Github no Travis-CI;

4) Adicionar o repositório do projeto no Travis-CI para ele executar os testes durante o *build*;

Com essas tarefas feitas, o build estará preparado para ação, executando todos os testes unitários a cada novo commit que for enviado para o repositório do projeto.

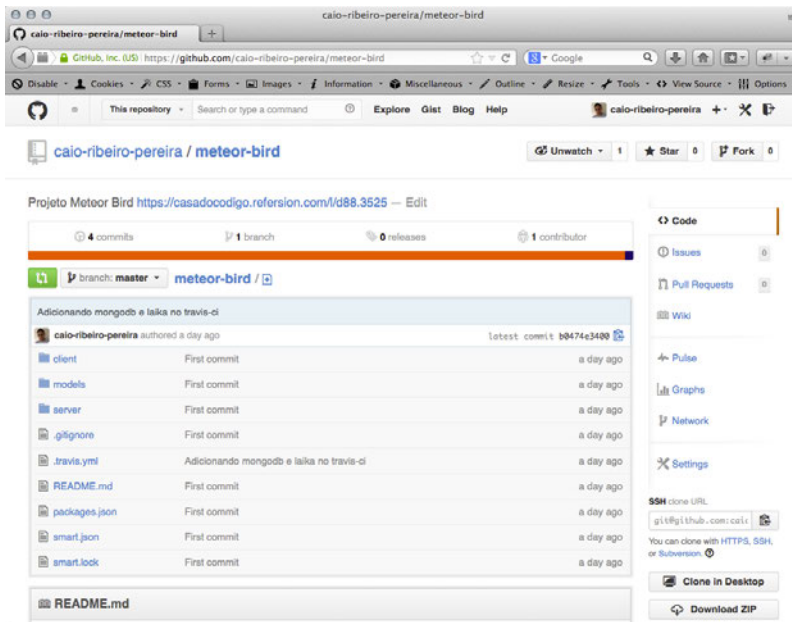



Figura 11.1: Repositório Github do Meteor Bird.

Figura 11.2: Rodando build no Travis-CI.

Figura 11.3: O build passou!

Capítulo 12

Preparando para produção

Se você chegou até este capítulo e conseguiu criar uma aplicação funcional e com testes, meus parabéns: você aprendeu os princípios e boas práticas do framework Meteor!

Neste capítulo vamos aprofundar em conhecimentos avançados, implementando no projeto alguns detalhes importantes para deixá-lo *ready for production*. Na prática, vamos focar nas seguintes tarefas:

- Configurar logs para monitorar alguns pontos da aplicação;
- Habilitar cache nos arquivos estáticos;
- Utilizar o package Fast Render para carregar templates mais rápido;
- Otimizar leitura no MongoDB usando o package Find-Faster;
- Trabalhar com variáveis de ambiente no projeto.

12.1 Monitorando a aplicação através de logs

Para monitorar o sistema através de *logging*, primeiro temos que configurar os pontos importantes do projeto para registrar um log do estado dos dados. Atualmente, não existe nenhum package compatível com Meteor, mas sim diversos módulos Node.js compatíveis. Como Meteor é construído em cima do Node.js, também temos a possibilidade de usar não todos, mas a maioria dos módulos Node.js existentes no NPM (*Node Package Manager*). Mas por que nem todos os módulos Node rodam no Meteor? O Meteor é uma plataforma que possui um contexto específico, que também trabalha fortemente integrado com banco de dados MongoDB. Exemplos de módulos Node que não serão compatíveis no Meteor são os módulos drivers de banco de dados: MySQL, Redis, SQL Server e outros. Outro exemplo de tipos módulos totalmente incompatíveis com Meteor são os *web modules*, responsáveis por criar uma aplicação web no Node.js, como **Express**, **Connect**, **Sails**, **Geddy** e muitos outros existente nesse link:

http://nodewebmodules.com

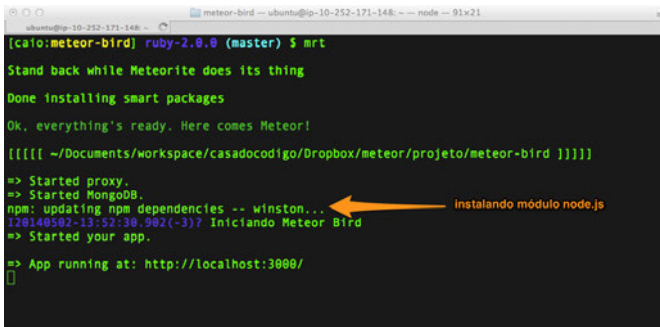
De fato, não faz o menor sentido adicionar um framework web Node.js no Meteor, visto que o próprio Meteor é um framework web. Por isso, nem todos os módulos Node.js serão compatíveis ou funcionarão corretamente dentro do ambiente Meteor. De qualquer forma, para os demais módulos compatíveis (que são muitos!), você consegue usá-los dentro de uma aplicação Meteor, e será esta a tarefa que faremos aqui.

Módulos Node.js, NPM e Meteor

Módulos Node.js dentro do Meteor já são utilizados desde a versão **0.6.x**. Além da incompatibilidade com alguns módulos devido ao contexto específico do Meteor, existe mais duas restrições do NPM no Meteor:

1) Os módulos Node.js somente funcionarão no *server-side* de uma aplicação Meteor;

2) Funções assíncronas precisam se converter para funções síncronas. Ao instalar o package `npm`, é liberada a função `Meteor.sync`, que permite executar funções assíncronas dentro de seu callback, convertendo-as para uma execução sincronizada.

Você pode ler outros detalhes sobre a integração do NPM com Meteor no blog oficial do Meteor através deste link:
https://www.meteor.com/blog/2013/04/04/meteor-060-brand-new-distribution-system-app-packages-npm-integration

Em vez de inventar a roda, vamos utilizar um módulo Node.js para gerar logs. Mas primeiro temos que habilitar o NPM em nosso projeto; para isso execute o comando:

```
mrt add npm
```

Com esse package instalado, crie no diretório raiz do projeto o arquivo `packages.json` (isso mesmo! Crie o `packages.json` no plural e não no singular, o que é muito comum em projetos Node.js.). Neste arquivo, vamos adicionar os módulos do NPM para utilizarmos em nossa aplicação. Para gerar logs, vamos usar o módulo `winston`, então edite o `packages.json` incluindo este módulo de acordo com código a seguir:

```
{
  "winston": "0.7.2"
}
```

Para que a instalação tenha efeito, apenas reinicie o servidor Meteor. Repare que, automaticamente, os módulos Node.js serão instalados quando se iniciar a aplicação.

Figura 12.1: Instalando dependência NPM ao iniciar o Meteor.

Agora vamos definir os pontos da aplicação que vão registrar logs. Você pode escolher qualquer local da aplicação desde que seja no lado do servidor para gerar um log. Para exemplificar seu uso, faremos algumas adaptações nos `models` do servidor para usá-lo.

Para carregar qualquer módulo Node.js, utilize a função `Meteor.require('nome_do_modulo')` e será isso que faremos para iniciar este módulo. Por default, o `winston` gera logs no console da aplicação, porém, em ambiente de produção, temos que gerar logs em arquivos para não perder as informações. Configurá-lo para gerar arquivos de logs é muito simples, basta adicionar um `winston.transports.File` como transporte e informar o local que será criado os arquivos de log via atributo `filename`. Veja em código como fazer isso, editando o `server/applications.js`:

```
winston = Meteor.require('winston');
winston.add(winston.transports.File, {
 filename: '../application.log',
 maxsize: 1024
});
Meteor.startup(function() {
```

```
 console.log("Iniciando Meteor Bird");
});
```

Com `winston` configurado, vamos utilizá-lo para monitorar os modelos da aplicação, primeiro fazendo um *refactoring* apenas na função `Post.publish` em `models/post.js`:

```
Post = new Meteor.Collection('posts');

Post.publish = function(message) {
 var currentUser = Meteor.user();
 var params = {
  message: message,
  time: new Date(),
  userId: currentUser._id,
  name: currentUser.profile.name
 };
 this.insert(params);
 winston.info("Post.publish: ", params);
};
```

Em seguida, adotaremos as mesmas convenções no código `models/friendship.js`. Para simplificar o exemplo, vamos gerar logs apenas nas funções `Friendship.follow` e `Friendship.unfollow`.

```
Friendship = new Meteor.Collection('friendships');

Friendship.follow = function(friendId) {
 var params = {
  userId: this.userId,
  friendId: friendId
 };
 this.insert(params);
 winston.info("Friendship.follow: ", params);
};

Friendship.unfollow = function(friendId) {
 var params = {
  userId: this.userId,
```

```
  friendId: friendId
 };
 this.remove(params);
 winston.info("Friendship.unfollow: ", params);
};
```

Com os logs implementados, você agora poderá visualizá-los via arquivo de log, que serão gerados no diretório `.meteor/local/build/programs/application.log`.


```
{
"0":{"message":"Hi!","time":"2014-03-22T18:38:19.547Z","userId":"Ptvb7H4PHbDZjf6CM","name":"Caio","_id":"3nNw6CLNJGMJqtDZ2"},
"1":{"message":"Muito legal esse Meteor Bird!!","time":"2014-02-27T22:55:23.445Z","userId":"Ptvb7H4PHbDZjf6CM","name":"Caio Ribeiro","_id":"yCKvoAFgeX9yRutb8"},
"2":{"message":"Topz!","time":"2014-02-27T22:55:02.755Z","userId":"Ptvb7H4PHbDZjf6CM","name":"Caio Ribeiro","_id":"yt6yNqX7RqgifTgLp"},
"level":"info",
"message":"Post.list: ",
"timestamp":"2014-03-22T19:02:11.061Z"
}
```


Figura 12.2: Arquivo de logs do sistema.

12.2 Habilitando cache em arquivos estáticos

Por default, o Meteor automatiza toda a tarefa de concatenação e minificação de código JavaScript e CSS quando a aplicação está em ambiente de produção. Isso já otimiza — e muito — a velocidade de carregamento dos arquivos estáticos da aplicação, além de evitar que façamos esse trabalho de automatização dessas tarefas. A única coisa que teremos que implementar é um controle de cache estático para garantir um carregamento mais rápido da aplicação e também evitar múltiplas requisições no servidor em busca desses arquivos.

Para habilitar o cache no Meteor, basta incluir o package `appcache`, que é um dos packages oficiais mantidos pelo *core-team* do Meteor. Execute o comando:

```
meteor add appcache
```

Feito isso, não se preocupe com mais nada: o próprio `appcache` se encarrega de fazer *caching* em todo o conteúdo estático.

12.3 Utilizando o Fast Render

Nesta seção, explicarei uma dica rápida, porém poderosa, cujo impacto é aumentar a velocidade da renderização dos templates. Para isso, vamos utilizar um recente package conhecido pelo nome **Fast Render**.

Basicamente, este framework melhora o carregamento inicial da aplicação, fazendo com que a velocidade de renderização dos templates seja 10 vezes mais rápido, sendo comparável a um carregamento SSR (*Server Side Rendering*). Outro ponto forte é que ele é integrado ao **Iron Router**, tornando possível configurá-lo no carregamento das rotas da aplicação.

Instale-o através do comando:

```
mrt add fast-render
```

Como nossa aplicação esta totalmente orientada às rotas, vamos apenas incluir um simples parâmetro (`fastRender: true`) nas rotas principais `'home'` e `'user'`, visto que elas são as únicas rotas que carregam templates. Edite o arquivo `client/lib/routes.js` e faça as seguintes alterações:

```
Router.map(function() {
 this.route('home', {
  path: '/',
  before: function() {
   var _id = Meteor.userId();
   this.subscribe("posts", _id);
   this.subscribe("friendship", _id);
  },
  data: function() {
   var _id = Meteor.userId();
   return {
    posts: Post.find({}),
    followers: Friendship.followers(_id),
```

```
    followings: Friendship.followings(_id)
   }
  },
  fastRender: true
 });

 this.route('user', {
  path: '/user/:_id',
  before: function() {
   var _id = this.params._id;
   this.subscribe("posts", _id);
   this.subscribe("friendship", _id);
   this.subscribe("isFollowing", _id);
   this.subscribe("user", _id);
  },
  data: function() {
   var _id = this.params._id;
   var isFollowing = Friendship.isFollowing(_id);
   Session.set('currentUserId', _id);
   Session.set('isFollowing', isFollowing);
   return {
    posts: Post.find({}),
    user: Meteor.users.findOne({_id: _id}),
    followers: Friendship.followers(_id),
    followings: Friendship.followings(_id)
   }
  },
  fastRender: true
 });
});
```

Com isso, já teremos uma grande melhoria visível aos usuários: um aumento na velocidade do carregamento dos templates.

12.4 Otimizando consultas no MongoDB com Find-Faster

Do mesmo criador do FastRender (o Arunoda Susiripala - @arunoda), recentemente ele lançou um package extremamente útil, que visa diminuir o tempo de leitura no MongoDB além de reduzir o uso de CPU do servidor. Como funciona? O driver MongoDB utilizado pelo Meteor esta constantemente convertendo objetos BSON (formato de objetos do MongoDB semelhante ao JSON do JavaScript) para objetos JSON. Esta tarefa consome muito a CPU, principalmente quando sua aplicação realizar múltiplas leituras em um mesmo registro. Neste caso o Find-Faster entra em ação, simplesmente fazendo *caching* desses objetos, evitando que essas leituras sejam realizadas no MongoDB.

Para instalá-lo execute o comando:

```
mrt add find-faster
```

Após sua instalação, basta substituir as funções `find` e `findOne` do MongoDB para as respectivas funções do Find-Faster que são `findFaster` e `findOneFaster`. Faremos essa pequena modificação nos dois modelos. Então edite o `models/post.js` atualizando a função `Post.list`.

```
Post = new Meteor.Collection('posts');

// Função: Post.publish...

Post.list = function(userIds) {
 return this.findFaster(
  {userId: {"$in": userIds}},
  {sort: {time: -1, name: 1}}
 );
};
```

Em seguida edite o `models/friendship.js` e atualize as funções: `isFollowing`, `timelineIds` e `followersAndFollowings`:

```
Friendship = new Meteor.Collection('friendships');
```

```
// Função: Post.follow...
// Função: Post.unfollow...

Friendship.isFollowing = function(friendId) {
 return this.findOneFaster({
  userId: this.userId,
  friendId: friendId
 });
};

Friendship.timelineIds = function(userId) {
 var timelineIds = this.findFaster({
  userId: userId
 }).map(function(f) {
  return f.friendId;
 });
 timelineIds.push(userId);
 return timelineIds;
};

Friendship.followersAndFollowings = function(_id) {
 return this.findFaster({
  $or: [{userId: _id}, {friendId: _id}]
 });
};

// Função: Post.followings...
// Função: Post.followers...
```

12.5 Configurando variáveis de ambiente

Por default, o Meteor reconhece e utiliza algumas variáveis de ambiente, que são **variáveis chaves**, responsáveis por fazer a aplicação utilizar um serviço externo do MongoDB, utilizar um domínio próprio, e também uma porta da rede diferente da porta padrão. Essas variáveis, respectivamente, se chamam `MONGO_URL` e `ROOT_URL` (em `ROOT_URL` é definido o **domínio + porta** da aplicação).

Você mesmo pode testar agora em seu ambiente de desenvolvimento, é

muito fácil. Primeiro, se estiver com a aplicação rodando, pare o servidor e, em seguida, inicie-o utilizando o comando:

```
ROOT_URL=localhost:5000 meteor
```

Repare que dessa vez a aplicação está rodando no endereço: http://localhost:5000

Este foi apenas um exemplo de como configurar um ambiente no Meteor. É possível utilizar customizações mais detalhadas através da variável global `Meteor.settings`.

Uma boa prática que podemos fazer agora é extrair as chaves `appId` e `secret` do Facebook, que usamos através da função `ServiceConfiguration.configurations.insert`. Para fazer essa extração, primeiro crie o arquivo `config/settings.json`. Neste arquivo, tenha o hábito de manter as importantes chaves e contas de serviços externos da aplicação, como por exemplo as chaves da autenticação via Facebook:

```
{
  "FB_SERVICE": "facebook",
  "FB_APPID": "INCLUIR APPID",
  "FB_SECRET": "INCLUIR SECRET"
}
```

Você também pode definir variáveis públicas que são expostas no *client-side* da aplicação. Para isto, basta incluir suas variáveis dentro da chave `public`, semelhante ao que podemos fazer no código a seguir:

```
{
  "FB_APPID": "INCLUIR APPID",
  "FB_SECRET": "INCLUIR SECRET",
  "public": {
    "FB_SERVICE": "facebook"
  }
}
```

Dessa forma, a variável `FB_SERVICE` está visível tanto no servidor como no cliente.

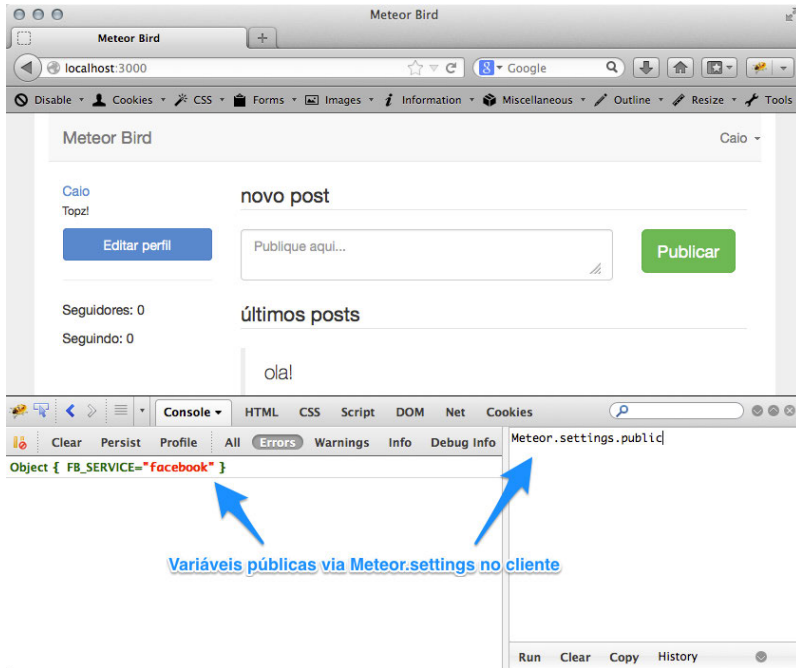



Figura 12.3: Usando Meteor.settings no cliente.

Depois de definir as chaves no `config/settings.json`, edite o código `server/lib/services.js` para que ele acesse os atributos de `Meteor.settings`:

```
ServiceConfiguration.configurations.remove({});
ServiceConfiguration.configurations.insert({
 service: Meteor.settings["FB_SERVICE"],
 appId: Meteor.settings["FB_APPID"],
 secret: Meteor.settings["FB_SECRET"]
});
```

Agora toda vez que levantar o servidor Meteor, será necessário rodar o comando:

```
mrt --settings config/settings.json
```

Como manter seguro o arquivo settings.json?

Manter a boa prática de guardar todas as chaves e senhas no `settings.json` garante sua centralização em um único lugar e facilita a gestão dessas informações importantes da aplicação.

Tome muito cuidado com esse arquivo! Evite deixá-lo exposto em ambientes públicos e, se possível, crie este arquivo no próprio servidor de produção para que somente a aplicação faça uso deste arquivo.

Assim você evita, por exemplo, enviar este arquivo através de um *commit* para seu repositório público do Github. Afinal, um repositório público com essas informações facilita para qualquer usuário mal intencionado invadir e manipular o seu sistema.

Com todas essas alterações feitas e rodando com sucesso, já temos uma aplicação que, além de funcional, já está preparada para ser enviada para um serviço de hospedagem em ambiente de produção. No próximo capítulo serão apresentadas algumas dicas para hospedar sua aplicação nas nuvens.

Capítulo 13

Hospedando uma aplicação Meteor

Parabéns! Se você sobreviveu até aqui, considere-se um vencedor! Afinal, caso não tenha percebido, você já desenvolveu seu primeiro projeto utilizando a tecnologia Meteor.

Para fechar com chave de ouro, neste capítulo vamos aprender algumas dicas sobre como empacotar um projeto Meteor para enviá-lo para um serviço de hospedagem nas nuvens.

No capítulo 2.6, foi apresentado como hospedar uma aplicação nos servidores gratuitos no subdomínio do `meteor.com`. O ambiente oferecido por eles é recomendado apenas para pequenas aplicações ou para hospedar versões betas, pois essas máquinas possuem um processamento muito fraco.

Na maioria dos casos, o recomendado é colocar nossa aplicação em servi-

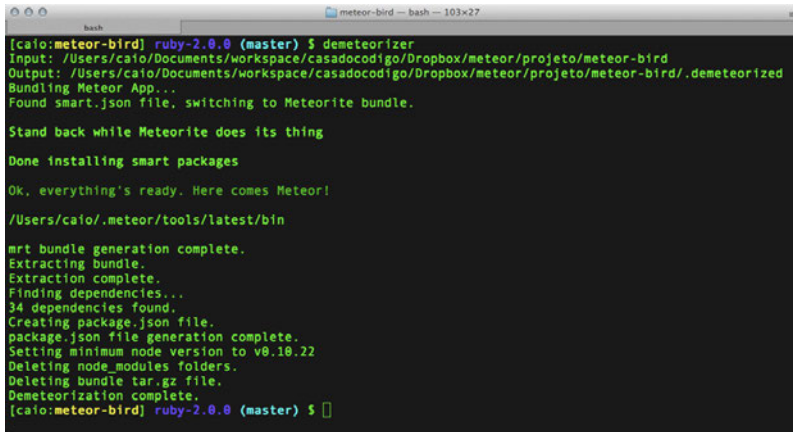
dores mais potentes, geralmente pagando por serviços dedicados, como por exemplo, a utilização de serviços nas nuvens que ofereça servidores para rodar aplicações Node.js e MongoDB (afinal, o Meteor é constituído por essas duas tecnologias).

Não entraremos em detalhes sobre como utilizar tecnicamente um serviço *cloud* e muito menos recomendarei qual é o melhor serviço para se hospedar uma aplicação Meteor. Porém, serão listados no final deste capítulo, os principais serviços compatíveis com Meteor.

13.1 Convertendo Meteor para Node.js com Demeteorizer

Algo muito interessante do Meteor é que temos a liberdade de hospedá-lo em **qualquer serviço com suporte a Node.js e MongoDB**.

Atualmente, é comum encontrar serviços com suporte a aplicações Node.js, o difícil é achar um serviço que traga um jeito fácil de fazer *deploy* em uma aplicação Meteor.

Para este caso, podemos converter um projeto Meteor para Node.js utilizando uma ferramenta chamada `Demeteorizer`.

Para instalá-la, basta rodar o comando:

```
npm install -g demeteorizer
```

Com isso, habilitamos no terminal um novo comando: `demeteorizer`.

Se você já possui um serviço de *host* compatível com Node.js, simplesmente configure as variáveis de ambiente `MONGO_URL` e `ROOT_URL`, de acordo com os endereços disponibilizados pelo próprio serviço de hospedagem. Se estiver com dúvidas, veja a documentação do serviço ou peça ajuda à sua equipe de suporte.

Com as variáveis de ambiente configuradas corretamente no ambiente de produção, execute na raiz do projeto o comando `demeteorizer`. Se não surgir problemas, o resultado será semelhante à imagem a seguir:

```
[caio:meteor-bird] ruby-2.0.0 (master) $ demeteorizer
Input: /Users/caio/Documents/workspace/casadocodigo/Dropbox/meteor/projeto/meteor-bird
Output: /Users/caio/Documents/workspace/casadocodigo/Dropbox/meteor/projeto/meteor-bird/.demeteorized
Bundling Meteor App...
Found smart.json file, switching to Meteorite bundle.

Stand back while Meteorite does its thing

Done installing smart packages

Ok, everything's ready. Here comes Meteor!

/Users/caio/.meteor/tools/latest/bin

mrt bundle generation complete.
Extracting bundle.
Extraction complete.
Finding dependencies...
34 dependencies found.
Creating package.json file.
package.json file generation complete.
Setting minimum node version to v0.10.22
Deleting node_modules folders.
Deleting bundle tar.gz file.
Demeteorization complete.
[caio:meteor-bird] ruby-2.0.0 (master) $
```

Figura 13.1: Rodando Demeteorizer.

Uma pasta com nome `.demeteorized` será gerada com todo o código-fonte do projeto em versão Node.js. Caso queria rodar a aplicação em sua máquina para conferir se a versão Node.js esta funcionando, basta rodar os seguintes comandos:

```
cd ./demeteorized
npm install
node main.js
```

Com essas dicas realizadas corretamente, basta enviar todo o conteúdo do diretório `.demeteorized` para o serviço de hospedagem e, por fim, iniciar o servidor Node.js no ambiente de produção, através do comando:

```
npm install
node main.js
```

13.2 Onde hospedar uma aplicação Meteor?

Há vários serviços Node.js com planos grátis e pagos. Segue uma lista com uma breve descrição de algumas plataformas populares:

Figura 13.2: Getup Cloud

É um serviço brasileiro (isso mesmo, temos alguém por aqui construindo nuvem também!), sua plataforma *open-source* é baseada no serviço OpenShift (http://openshift.com) , infraestrutura Amazon e cobrança por hora e *on-demand*. Além de rodar Node.js, também suportam outras linguagens (Java, PHP, Ruby, Python, Perl), frameworks (Ruby on Rails, CakePHP, Django, Wordpress) e banco de dados (MySQL, PortgreSQL e MongoDB). O mais legal é que a plataforma oferece escalabilidade automática para as aplicações, criando e destruindo máquinas de acordo com o volume de conexões simultâneas, sem que você precise entender de *load balance* ou outros detalhes técnicos.

A administração pode ser feita pelo CLI através do comando `rhc` ou pelo web admin do próprio site, e o deploy é feito através do git.

As aplicações rodam em instâncias chamadas de *gear*, que é uma máquina na nuvem. Quando os *gears* são criados, eles recebem uma URL no formato http://nome_app-namespace.getup.io. Eles permitem incluir certificado SSL e também o uso de um domínio próprio. O formato de pagamento é através

de paypal, cobrança por hora e em reais.

Site oficial: http://getupcloud.com

Figura 13.3: Modulus

O Modulus é um serviço PaaS focado em Node.js. Um dos seus diferenciais é que ele também suporta nativamente o Meteor. De banco de dados você terá disponível apenas o MongoDB, e caso sua aplicação funcione em real-time, você poderá usar o protocolo WebSockets. Nele é possível customizar domínio e certificado SSL. A versão trial oferece 15$ de créditos para colocar uma aplicação de pequeno porte em cima de um servidor com 64 MB de *database* e 1 GB de *storage*.

Site oficial: https://modulus.io

Figura 13.4: Heroku

Conhecida pelos programadores Ruby, Java, Python e Closure, e principalmente o Node.js. O Heroku possui uma CLI para administração de aplicações e eles oferecem gratuitamente o domínio `nome_app.herokuapp.com` ao criar uma aplicação. O mais interessante desse serviço é que eles possuem diversos *add-ons* que permitem a integração de serviços *third-party* em sua aplicação. Um exemplo disso são o *add-ons* que se integram com o Twitter, serviços da Amazon, serviços de banco de dados e cache, plugins para serviço de mensageria SMS e muito mais.

Nem todos os *add-ons* são gratuitos, muitos deles são oferecidos como serviços adicionais a serem pagos. O pagamento também é *on-demand*, ou seja, você paga pelo que usar ou paga pelo *add-on* que instalar.

O deploy é efetuado através do git e toda a administração da aplicação pode ser feita tanto pelo seu web admin como pelo seu CLI, via comando `heroku`.

Site oficial: http://heroku.com

Figura 13.5: Nodejitsu

Com a proposta de um serviço de hospedagem em nuvem, o Nodejitsu traz consigo diversos módulos para desenvolver aplicações Node.js em sua plataforma. Esta é uma plataforma 100% Node.js que permite tanto a hospedagem de aplicações web, como a hospedagem de módulos privados para Node.js. Possui uma documentação bastante clara e didática sobre sua API e oferece gratuitamente um domínio para sua aplicação através do subdomínio `nome_app.nodejitsu.com`. Além disso, também possível o redirecionamento de sua aplicação para um domínio próprio.

Eles oferecem suporte ao WebSockets, suporte aos banco de dados Redis, CouchDB e MongoDB. Toda gestão de uma aplicação e deploy é feita através

do seu próprio CLI, que é o comando `jitsu`. Este é um serviço cloud totalmente pago e dedicado inteiramente para o Node.js — não oferece nenhum plano gratuito.

Site oficial: https://www.nodejitsu.com

Capítulo 14

Como organizar um projeto Meteor

14.1 Convenções de diretórios e arquivos

Algo muito interessante do Meteor é como ele estrutura e organiza seus códigos, ou seja, suas convenções de diretórios que, quando bem utilizadas, fazem com que os módulos sejam carregados corretamente em seus respectivos contextos. Dominar esses conceitos o poupará de futuros problemas em relação a módulos que não executam corretamente em seu projeto.

Recomendo que entenda o significado de cada diretório e também entenda sobre que tipo de módulo se deve criar neles. Veja a seguir as principais convenções de nomes de diretórios e arquivos:

- `lib`: inclua bibliotecas que serão carregadas tanto no cliente como no

servidor. Um bom exemplo são códigos de regras de negócio compartilháveis entre cliente e servidor.

- `lib/environment.js`: Javascript de configurações gerais sobre ambiente.
- `lib/methods.js`: definições de rotinas para o `Meteor.method`.
- `lib/external`: bibliotecas externas de outros autores, exemplo: plugins do jQuery.
- `models`: definições de modelos que representarão collections do MongoDB.
- `client/lib`: códigos específicos para o cliente. Todas essas *libs* serão carregadas primeiro, ou seja, antes dos demais códigos dentro da pasta *client*.
- `client/lib/environment.js`: configurações de pacotes para o cliente.
- `client/lib/helpers`: pasta dedicada para helpers e events dos templates.
- `client/application.js`: código de inicialização no cliente; geralmente, utilizam-se funções de *subscriptions* ou *rotinas internas* a serem executadas como callback da função `Meteor.startup`.
- `client/index.html`: HTML principal.
- `client/index.js`: Javascript principal que carrega primeiro no load da página. Ele também pode ser chamado de `main.js`.
- `client/views/nome_do_template.html`: template HTML (mude `nome_do_template` para o nome real do template).
- `client/views/nome_do_template.js`: código Javascript para o respectivo template (mude `nome_do_template` para o nome real do template).

- `client/views/nome_do_sub_template`: você também pode criar subtemplates, geralmente conhecidos como *partials*, e são trechos de templates utilizados em um outro template.
- `client/stylesheets`: diretório CSS, no qual você pode colocar código CSS, Less ou Stylus, e todos eles são automaticamente compilados e minificados.
- `client/stylesheets/application.styl`: caso trabalhe com CSS modular, recomendo que crie este arquivo e, dentro dele, insira os `@import` de seus módulos do *Stylus*.
- `client/stylesheets/application.less`: segue o mesmo conceito da dica anterior, mas é para a utilização do compilador *Less*.
- `server/application.js`: código de inicialização no servidor; geralmente, utilizam-se funções a serem executadas como callback da função `Meteor.startup`.
- `server/publications.js`: definições de rotinas para o `Meteor.publish`.
- `server/lib/environment.js`: configurações de pacotes para o servidor.
- `packages`: mantém packages dependentes do projeto, que são fornecidos pelo Meteorite.
- `public`: pasta de arquivos estáticos. Adicione nele imagens, fontes e outros arquivos que serão servidos estaticamente.
- `tests`: diretório para criação de testes unitários. Por default, nenhum código aqui é carregado ao levantar sua aplicação Meteor. Esta é uma pasta dedicada para frameworks de testes.

Capítulo 15

Continuando os estudos

Obrigado por ler este livro, para mim é uma honra tê-lo como leitor! Se você chegou aqui é porque o livro acabou, e espero ter passado conteúdos de grande utilidade para seus estudos sobre o Meteor. Como tudo no mundo da programação esta sempre evoluindo, listarei alguns sites relevantes para você continuar antenado nesta tecnologia.

Sites, blogs, fóruns e slides

- Site oficial Meteor — http://meteor.com
- Documentação oficial — http://docs.meteor.com
- Documentação não-oficial — https://github.com/oortcloud/unofficial-meteor-faq

- Atmosphere — https://atmospherejs.com
- Site oficial Node.js — http://nodejs.org
- Documentação Node.js — http://nodejs.org/api
- Site oficial MongoDB — https://www.mongodb.org
- Documentação MongoDB — http://docs.mongodb.org
- SQL to MongoDB — http://docs.mongodb.org/manual/reference/sql-comparison
- Google Meteor Brasil — https://groups.google.com/forum/#!forum/meteor-brasil
- Facebook Meteor Brasil — https://www.facebook.com/groups/meteorbrasil
- Underground WebDev — http://udgwebdev.com
- NetTuts+ — http://net.tutsplus.com
- MeteorHacks — http://meteorhacks.com
- Meteor, um overview sobre a plataforma — https://speakerdeck.com/caioribeiropereira/meteor-um-overview-sobre-a-plataforma
- Blog Manuel Schoebel — http://www.manuel-schoebel.com/blog

Eventos, screencasts, podcasts e cursos

- EventedMind — https://www.eventedmind.com
- CodersTV — http://coderstv.com
- Meteor Podcast — http://www.meteorpodcast.com
- Meetup Meteor SP — http://www.meetup.com/Meteor-Sao-Paulo
- Ebook DiscoverMeteor — http://www.discovermeteor.com

- Ebook TestingMeteor — http://testingmeteor.com

Que essas referências sejam de grande utilidade. Obrigado por ler este livro!